Technical Design Document

Mud, Sweat, and Gears

Press Start and Interplay Productions

Prepared by Press Start for Interplay Productions
11/04/99

version 1.01

Introduction 7

Technical Overview 7

Memory Map 7

Staff 8

Contact Information: 8

Localization Considerations 9

Derivative Game Considerations 9

Portability considerations 9

Development Tools 9
Art 9
Programming 10

Major Components 10

Scene Manager 11

Scene Manager Goals 12

Major Components 12
Core processing scheduler 12
Control Node Tree 12
Gem Tree 12
Template Tree 14
Instance Tree 14
Controller Tree 14

More on Gems for Space Partitioning 15

Scene Manager Task Processing 17

Animation 18

Animation Data 18

Geometry animation 18

Light attribute animation 18
RGB color 18
Ambient lighting (animation) 18
Intensity multiplier 19
Umbra & Penumbra 19

Object attribute animation 19

Event spawning animation 19

AI/Controllers 20

Car AI 20

Terrain & Deformation Controller 20

A further discussion of AI and our approach to implementation 21

Master or Slave? 21

AI is the master, physics is the slave 22

Physics is the master, AI is the slave. 24
Best of both worlds: The solution 25
Support AI types 26
Triggers 26
Event triggers 26
Timed triggers 26
Collision triggers 26
Swappable objects 26
Scenery generators 27
Renderer 26
Requirements for a Renderer 28
Object Description 28
The Conceptual 3D pipeline 29
Lights, Shadows and MultiPass Rendering 29
More on lighting 30
Reflection Vectors, Environment Mapping 31
Transparency and Translucency Means Sorting 32
Levels of Detail (LODs), Bounding Calculations and Interpolation 32
Progressive Meshes Reduce LOD Data 32
Collisions Need to Be Computed in World Space 33
Z Problems and Solutions 33
Surface Deformations and Displacement Maps 33
Fog and Clouds 33
Reflections 34
Making Water Look Alive 34
High Level Renderer API 34
Shadows 35
Shadow Volumes 36
Dynamic shadows 36
Camera System 37
Definitions 38
General CS Functions 38
Offset Functions 41
Locking Functions 41
Miscellaneous Functions 42
Secondary Control Functions 42
Physics & Collision 43
Brief Overview 43
Physics System 43
Collision Detection 43

Vehicle model 44
Engine vibration 44
Suspension system 44
Body Sway 45
Tires and Road Friction 45
Flat Tires 45
Bumpers 45
Oversteer and understeer 46
Engine Power and Differential Traction 46
Collision detection 46
Collision deltas, or Yes, We Really Do Have to Check That Often 47
Rollover and the problem of self-righting 48
Factors that aren't in the model 48
Performance Analysis 48
Some math 49
Pertinent Structures 50
Networking 52
Brief Overview 52
Goals 52
Block Diagram 52
Description of Figure 1 53
Third party software – communication SDK 53
Transport layer 53
Low-Level API 53
Safe connection 54
Fast connection 54
Fast connection with error correction 54
Low-level API usage 55
High-Level API 56
Object model of high-level API 56
Client-server architecture 57
Server component 57
Server's arbiter functionality (sample for two clients) 58
Client component 58
Common 59
Room level 59
Object level 60
Menu/UI Implementation 63
Adding Menu AI to an object in a scene 63

Controlling a menu during run time 63
Overlaying a menu 64
Taking control of the player's game controller 64
Passing menu selections back to the engine 64
Menu object text 64
Input Handling 65
Art Tool Chain 66
Brief overview 66
Calliope™ plug-in 66
Goals of the tool set 68
Block Diagram 68
Overall description 69
Third party software 69
Project developed software 69
Calliope™ – art tool plug-in 70
Gems 70
Hierarchy Inhibitors 70
Collision Spheres 70
LOD Minimum Vertex List 70
Animation 70
Position, Rotation, Scale 70
Visibility 71
Text "Animation" Events 71
Formatted Structured Events 71
Lights 71
Sound Events 72
Plug-in code 74
Gems (Space Partitioning) 74
Model Templates 75
External Data files 78
AI Templates 78
Particle System 80
Brief overview 80
On particle systems ... 81
Emitter 81
Canonical Particle 81
Particle System Parameters 82
The Particles 84
Interface to the Particle System 84

Texture cache 86
FMA 87
Normal scene data 87
FMA director 87
FMA cameras 87
FMA actors 88
Golems 88
How FMA actors move golems 88
FMA Compliancy 88
Starting an FMA 89
Memory Card 90
Virtual File System 91
Memory Management 92
DRAM Management 92
SPU-RAM Management 92
VRAM Management 92
DRAM/SPU-RAM Memory Manager API 93
Geometry Compression 94
On higher order surfaces 94
Various methods to reduce geometry in DRAM 94
Instant Replay 95
Special Effects 96
Lens effects 96
Lens glare 96
Lens Flare 96
AI Support 96
Generic AI controller 97
Particle Effect AI 97
Appendix A 101
Right Threaded Binary Trees and Tree Traversal 101

Introduction

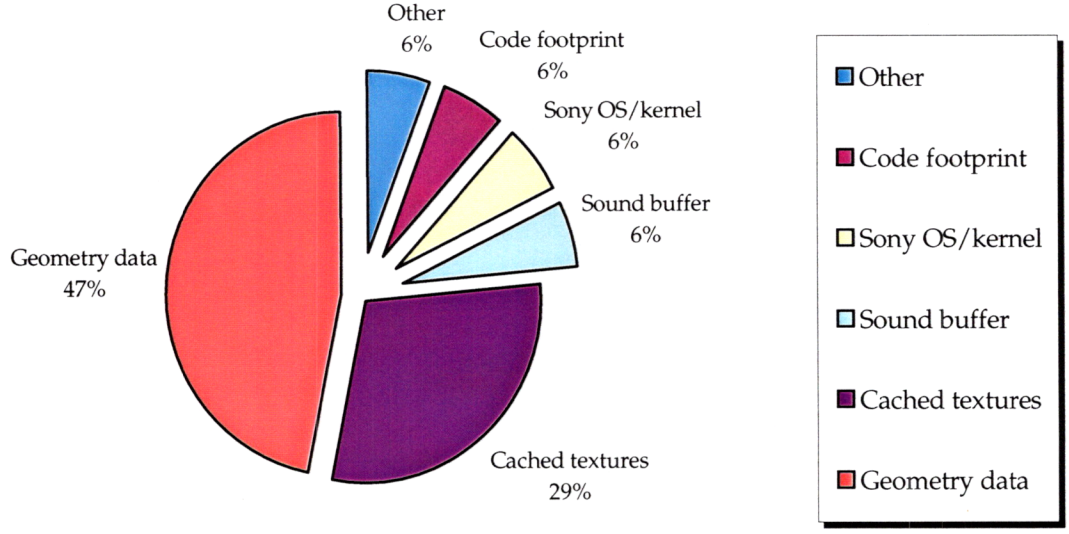
Mud, Sweat and Gears (MSG) is an off road racing game in which the player battles opposing vehicles and extremely challenging terrain – including moving ice, falling boulders, tornados, and treacherous jumps – to be the first to the finish. The game is designed for next generation consoles (specifically, the Sony PlayStation 2), and will take advantage of the next level of hardware to provide dynamic lights, real-time terrain deformations, realistic physics, high use of particle effects, high amounts of animated models, shadow effects.

Technical Overview

Platform	– PlayStation 2 (PS2)
Target Frame Rate	– 60 fps
Target Gold Master Candidate	– August 7, 2000
Target Release Date	– September 15, 2000

Memory Map

Sony OS/kernel	2mb	
Geometry data	16mb	
Cached textures	10mb	
Code footprint	2mb	
Sound buffer	2mb	(SPU, off DRAM)
Physics data	8k	
Scene Manager (including data for 5000 objects)	~128k	
AI data	16k	
Terrain AI data	32k	
Instant Replay	512k	
Approximately 1 1/2 mb left over for expansion		



Staff

Press Start, Management

Paul Shaw	President
Ross Horman	Project Manager, Producer
Jim Tomasko	Technical Director, Engineering Lead
Scott Paterson	Art Director

Press Start, Engineering

Jim Tomasko	Scene Manager, animation, FMA, sound & music
Carla Meninsky	Rendering API, high level renderer, AI, terrain
Carolyn Wales	Physics, collisions, particle systems
Doug Coward	Menu systems, fonts, UI, input, binary tools
Alexey Kadukin	Tool chain, networking, memory management

Press Start, Art

Scott Paterson	Art Director, animation, modeling, storyboards,
Bill Yeatts	Modeling, animation, textures, storyboards
Sean Spitzer	Environment modeling, textures, animation, storyboards
Scott Kinkaid	UI, production work, textures, storyboards
Lee Steg	Illustration, design, modeling, textures, storyboards
Yu Gu	Object modeling, animation, textures
Paul Shapiro	Object modeling, animation, textures

Interplay, Management

Trish Wright	Vice President of Product Development
Jim Molitor	Executive Producer
Shawn Jacoby	Producer
David Sullivan	Technical Director
Rob Nesler	Art Director

Interplay, Programming

John Morgan	Low level renderer

In addition to the low level renderer, Interplay is responsible for supplying the project with all sound assets (music, sound effects, voices) necessary for the creation of the game.

Contact Information:

Paul Shaw	408-962-1004	pshaw@pressstart.com
Jim Tomasko	408-962-1006	jtomasko@pressstart.com
Scott Paterson	408-962-1016	spaterson@pressstart.com
Ross Horman	408-962-1005	rhorman@pressstart.com
Jim Molitor	949-223-5838	dsullivan@interplay-usa.com
Shawn Jacoby	949-223-5700	sjacoby@interplay-usa.com
David Sullivan	949-223-5736	dsullivan@interplay-usa.com
John Morgan	949-223-5736	jmorgan@interplay-usa.com
Rob Nesler	949-223-5824	rnesler@interplay-usa.com

Localization Considerations

Some of the text is embedded art (such as title screen logos and the like). Any of this text that requires localization changes will require art rework.

All other text will reside in a text file where all the text is indexed and can be changed easily. The ability for the text "boxes" to resize due to expanding word size will be used when possible. Character strings used for display purposes will be stored as DBCS wide strings (UNICODE).

Frame rate timing for animation is done in floating point, and synchronized with time instead of frames per second, so conversion to PAL will be primarily an issue of screen resolution – provided that PAL PS2 units output a different resolution than their NTSC counterparts.

Derivative Game Considerations

Various parts of the game and tool chain are built to be reusable, especially for the task of making games based on 3D worlds. The 3D Studio MAX plug-in is built to be extensible for building 3D worlds and characters. Much of the specifics required to make a racing game are in configuration files that may or may not be reused at the end of MSG. The same design considerations have been included in the engine design to get reuse from the scene manager, memory manager, math library, error management, input management, sound, collision detection, timing, and API to the renderer.

Portability considerations

The majority of the core engine and Calliope™ plug-in for MAX have been designed with portability in mind. Areas such as the input manager and renderer have high level API's which allow ease of use in making all but the low level portions of them portable to other platforms (such as the PC) at a later date.

The binary converter (which converts the MAX plug-in output to a format the engine may work with) is a candidate for rewrite for other platforms, as well as the actual geometry (polygon and texture data) format within the engine. Care has been taken to fully separate the actual model data format from the model's "skeleton" and animation data to insure that geometric data is contained in a localized area of the engine code.

One last major portability concern is the actual amount of model and AI data, (game specific vs. engine specific). If portability of MSG (as a game and not as an engine) is an issue in the future, the current plan is to add SKU specific tags on all the model data in the MSG 3D Studio files, where the artist may tag objects to be exported for one format and not another. This allows the artist to create a scene in which all objects are present for one platform, with some objects not present on others – sort of a platform implementation of LOD.

Development Tools

Art

3D Studio MAX R3	– modeling, world building, animations, texturing
Maya 2.5	– sculpting of primary terrain regions
Animatek's World Builder	– modeling of trees and other terrain elements.
Adobe Photoshop	– creation of textures
Debabelizer Pro	– animated textures, super-paletting

Programming

MS Visual Studio 6.0	– tools, game and engine programming, at least until we have PS2 development systems.
MS Visual SourceSafe 6.0	– source control and asset management (If we end up doing PS2 development under Linux we may have to change to Perforce or another version control system)

PS2 development will be done with either the Sony supplied debugger/complier (GCC and GDB), the Metrowerks PS2 IDE, or the SN Systems PS2 IDE. Press Start intends on working entirely within a Windows environment unless the PS2 development tools for Windows are not in an acceptable condition. If the Windows tools are not acceptable, we will migrate to the most acceptable solution.

Major Components

Scene Manager	Jim Tomasko
Animation	Jim Tomasko
AI (including terrain deformation)	Carla Meninsky
Renderer	Carla Meninsky and John Morgan
Camera System	Doug Coward
Physics	Carolyn Wales
Collision Detection	Carolyn Wales
Networking	Alexey Kadukin
Audio and Data Streaming	Jim Tomasko and Carolyn Wales
Menus and UI Toolkit	Doug Coward
Input Handling	Doug Coward
Low Level Hardware Interface	Carolyn Wales and Carla Meninsky
Art Tool Chain	Alexey Kadukin, Jim Tomasko, and Doug Coward
Particle System	Carolyn Wales
FMA	(Jim Tomasko
Memory Card	Doug Coward
Virtual File System	Carolyn Wales
Memory Management	Alexey Kadukin
Geometry Compression	Carla Meninsky
Sound & Music	Jim Tomasko
Instant Replay	Doug Coward
Special Effects	Entire Technical Team

Scene Management

Scene Manager

The Scene Manager is responsible for running the main game loop, for determining basic LOD information needed for object processing, and for broadcasting any important messages (such as, "restart game", or "stop game") to the objects in the game.

The main game loop is divided into five different time slices:

- Get current time
- Poll the input devices
- Do AI[1]
- Do animation[2]
- Render the scene

Each object that has an AI (i.e., can "do" something) has a controller that is registered with the Scene Manager. The controllers are arranged in hierarchical order corresponding to the hierarchies within the objects being controlled. For instance, a gun attached to a person's hand would have a controller that is a child of the parent controller. Please note that the things these controllers manage need not be geometric objects. There are controllers for sounds, cameras, lights, triggers, and anything else that has AI processing.

The controllers contain, among other things, pointers to functions used for processing messages from the Scene Manager - the "Do AI" of step 3 is such a message. Since not all controllers need to be processed every loop, the top level of the hierarchy contains priority nodes which act as gateways to the controllers below. In general, the priority nodes contain counters and will allow their controllers to be processed every *n* times through the loop.

The animation time-slice is also controlled through the controllers. The Scene Manager walks through the list of controllers, telling them to do their animations. Note that the child controllers are inhibited from the animation time-slice as the actual objects they point to are already hierarchically linked and will animate together naturally.

The AI time-slice determines if an animation is playing and how much time to take when transitioning between animations, but the animation time-slice is responsible for setting up the matrices for that frame of animation, and interpolating between matrices if transitioning between two animation sequences. When the animation slice determines that the animation has finished, it sends a message to the through the controller notifying the AI.

When the frame rate is threatening to drop below the target frame rate (or already has), the Scene Manager sets a flag that notifies the various controllers. The individual controllers and their AIs are responsible for doing whatever is necessary to keep the frame rate up. At this point it is premature to say exactly what we might do, but options include switching to a lower level of detail for non-critical objects, running the physics and collision engines less frequently, or simplifying processing of objects that are off screen.

---

[1] The term AI (above) is used in a general sense. Essentially, each object is given a chance to do whatever it feels is appropriate. That may be camera positioning, physics, collisions, setting which animation to play, and standard AI type tasks. AI for an object in the world is intended to generally be set up as a state machine, but there is no limitation imposed that requires this.

[2] Each object has an opportunity to set itself up for the next frame of animation, if it is animating.

Risks - Low.

The Scene Manager is relatively straightforward. Creation of the space-partitioning gems is potentially tedious, but is not a major risk in the game. The only real risk (and it is slight) is that the structure of the Scene Manager might prove too inflexible to support some of the real-time callback and interrupt functions that we need.

Scene Manager Goals

The scene manager has 5 primary goals:

- Get the current project out on time as full featured as possible.
- To allow a very high level of re-use for not only possible sequels, but also for other games based upon 3D world environments. Every attempt has been made to insure that the scene manager only suffers from two dependencies, one is that it offers an operating system style environment for management of 3D worlds only; The other is that the 3D worlds take advantage of a particular space subdivision technique which is described later. The scene manager contains API calling prototypes which indicate when to render, when to get game controller data, along with other such calls, but does not actually perform these functions.
- To allow a small but powerful set of core code that performs in a consistent manner, allowing the major sections of the game to be reused with the goal of increasing reliability.
- To allow for rapid prototyping of future games.
- To be easy to understand and use.

The scene manager is akin to the operating system of the game engine. This document attempts to detail not only the inner workings of the scene manager, but also attempts to stress the importance of a thoughtful integration of data and functions that will be added to produce a game with it.

Major Components

The scene manager is composed of the following components:

Note: Appendix-A describes the tree mechanism, as well as a generalized traversal method.

Core processing scheduler

The scheduler may also be simply called the "main loop". The scheduler is a single tasking poling loop which determines when to perform each of the main tasks, which are simulation time synchronization, game controller retrieval, AI processing, animation positioning, camera processing, and rendering. Each object in the world is handled through a callback mechanism which receives messages indicating what task they are to perform.

Control Node Tree

Each object in the world that is dynamic (any object that may be moved or has a "brain") has a controller associated with it, and registers that controller with the controller tree. The controller tree is used to schedule AI processing and animation time slices. The controller tree is also used to send general broadcast messages to all controllers (such as game over).

Gem Tree

The gem tree is a hierarchical database of nodes that denote the space partitioning of the scene (see *Gem Space Partitioning*). Each leaf node (the nodes that have no children) denotes a 3D volume of space that may contain objects in the 3D world. Breaking up the scene into these small manageable pieces allows

the engine to focus its attention on specific areas of interest relevant to the current task it is attempting to accomplish.

Gems offer great advantages over other forms of space partitioning such as BSP trees. Although gems require time from the artist to hand build (using the Calliope™ plug-in), the time spent is recouped in the export speed of gem-based scenes. The engine also gains many speed benefits from a gem-based system over other forms of space partitioning. The following lists some of the benefits unique to gem-based space partitioning.

Export Speed

The export speed of gems is significantly greater than BSP trees. While BSP export times are measured in hours, gem-based exporting is measured in seconds or low minutes. This is due to gems are built by the artist, and no time intensive ray-tracing algorithms are required. Admittedly, the artist has to build the gems in the scene, but the time is made up in the project by allowing the artist to make rapid changes to a scene for life of the project. The static geometry in each region is further subdivided into oct-trees. This provides an even finer level of object culling that is not time intensive to export as each static polygon is merely tested against an axis aligned cubic volume, which is another quick process.

Visibility lists

The artist defines which gems are visible to other regions. This offers a unique set of benefits not found in other space partitioning schemes:

- Rendering candidates are reduced to the gem containing the camera and the gems in that regions visibility list.
- Collision checking of any object is bound to other objects in the gem the object is in, and the gems visible to the object's home gem. This removes the typical requirement of distance-checking each object in the scene to every other object, or coming up with special-case dependencies on collisions to reduce the amount of checking in the world.
- AI "level of detail" requirements may be tailored on an object-by-object basis depending on if the object is in a visible area or not.
- Real time lighting tests are reduced to checking only the lights within the visibility lists, and then only lighting those gems.
- Processing time requirements based upon gems and their visible gems are localized for easy testing, which allows fine attention to detail of each area's look. The artist is much more free to make decisions on what is important in each area... balancing special effects and object AI to the amount of polygons in an area.
- Far away objects, such as distant mountain peaks, may be tagged visible from any logical place in the scene.

Environmental parameters

Gems contain environmental parameters to define attributes local to the space defined by the gem. These environment parameters are configurable from game to game.

In MSG the parameters are:

- Ambient light
- Sound reverberation constants
- Sound dampening
- Vehicle attribute bonus/detriment (such as cursed areas in the Voodoo Village)

Template Tree

The template tree holds templates for all objects that are either currently in the world, or that need to be kept around to make instances on the fly (such as explosion effects, where there is no time to get them from the storage media in time to use them).

It contains the node hierarchies of each object that's either currently instanced in the world, or objects tagged to not remove their templates when the last one is destroyed. The tree consists of one root, and each direct child of the root is a root object of a sub-tree which describes the hierarchy of a multi-node object. The nodes in the template tree contain pointers to non-dynamic data, such as the geometry for an object. If the object requires to modify it's geometry, it must create a copy of the template that is unique to it. The template also points to all the animation data per node, and all other read-only data.

Instance Tree

The instance tree holds the objects that are instanced in the world. Any data that is common to all instances of a template merely point back to it's template to save space. Only data such as an instances unique parameters, where it is in the world, it's current animation state and the like are contained in the instance.

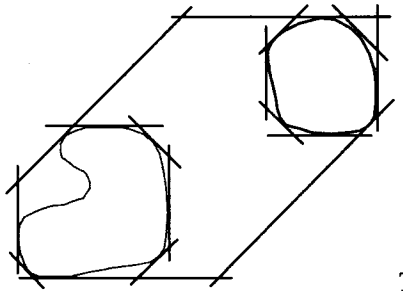
It contains a root object, where each of it's children are the root level object of an instance in the world. Note that instances may be linked to each other. This is shown in the block diagram where the instance made from the template with 2 nodes is linked to the instance with 3 nodes. One application of this would be a bird that has been hit by a moving car, and is trapped in the car's bumper. This is done by linking the bird to the car so that as the car moves the bird goes along for the ride.

Controller Tree

The controller tree contains a root object, where each of it's direct children are priority control objects. The children of the priority control objects are the AI controllers for each instance that is dynamic in any way. Controllers are the objects that receive AI time slices. When a controller is evoked to perform it's AI, it has a pointer back to the instance data so that it may manipulate the geometry of the instance (such as move it to a new location in world space). The priority nodes are used to manage processing load, and are configured on a game-by-game basis. One scheme is to process the priority one nodes each frame, while processing the next priority every 10 frames, and never processing the last priority. Another scheme is process every frame (there's not a terrible amount of controllers in a typical game), but to use the knowledge of the priority to perform a lower level of AI, such as in a race game the software may move cars out of the camera's view to lower priority controllers, dependant on how far away it is. At the lowest priority the AI might just make a few simple predictions on where the car should be and just warp it there.

More on Gems for Space Partitioning

Kay and Kajiya in SIGGRAPH 1986 introduced bounding volumes, which fitted the objects as tightly as possible, reducing the "void" volume. In their method the bounding volume consists of **pairs** of parallel planes, made to fit convex hulls of objects tightly. This is illustrated below:



Two-dimensional representation only.

Intersection testing is a simple generalization of the intersection box testing. In 2D a minimum of 2 pairs of "planes" (lines) is needed, in 3D at least 3 pairs are required. The number employed depends on the desired tightness of fit.

In our implementation, space is partitioned hierarchically into regions known as gems, 18 sided 3-dimensional objects whose sides are aligned to one of the major axis, or 45 degrees off from a major axis.

Each gem has a list of objects within it, as well as a list of other gems that are visible from the current gem. When it comes time to render, the Scene Manager assembles a visibility list of objects based upon visible gems, and sends this information to the high level renderer.

Gems are created by the artist or production assistant (using the Calliope™ plug-in tool that is integrated with 3D Studio MAX). They can be any number of pairs from three (a six sided Gem) to nine. A careful construction of Gems can greatly reduce the amount of processing overhead.

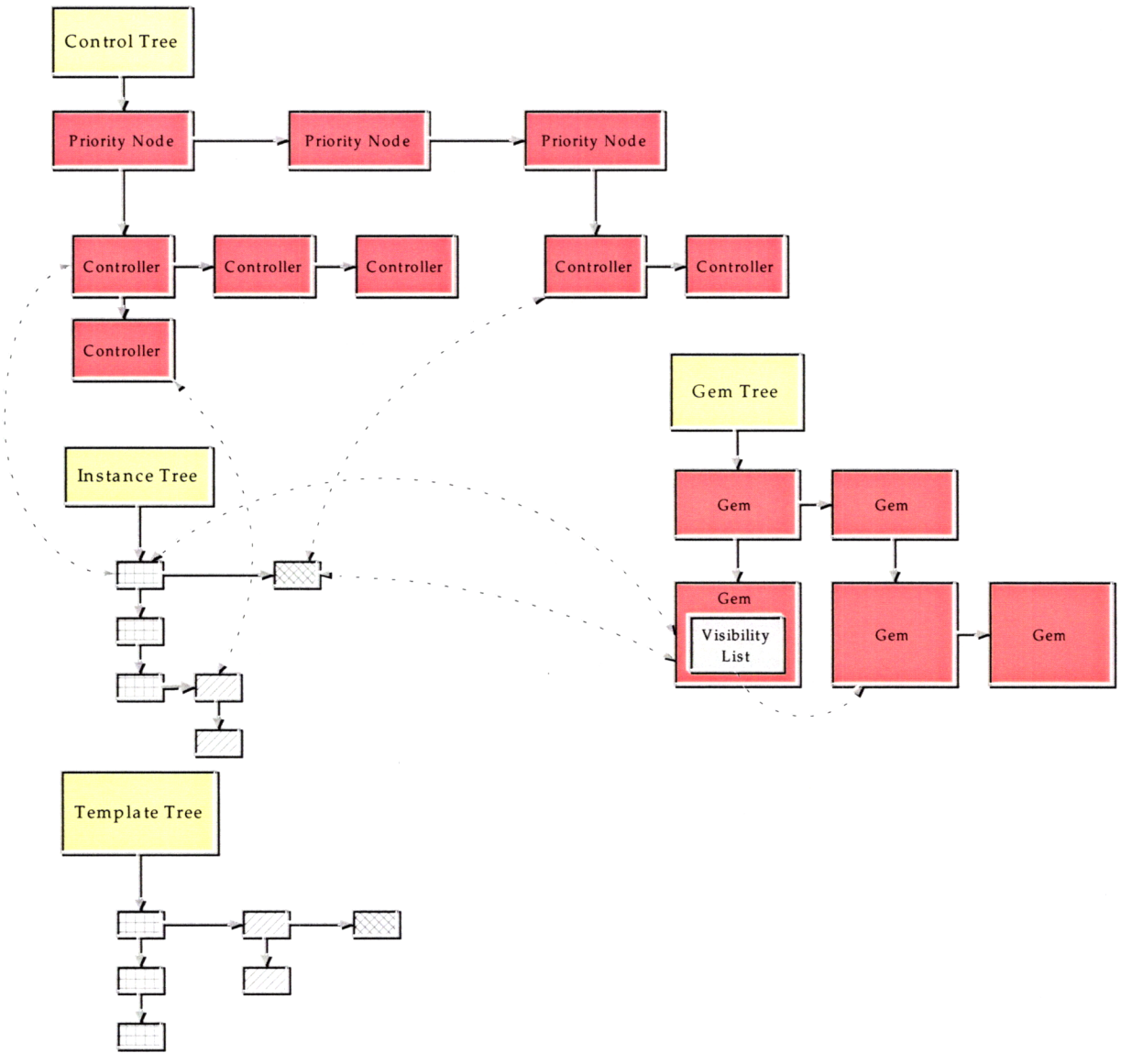
Options in the Calliope™ plug-in for Gems control are as follows:

- Force creation of and use six-sided Gems for contiguous rectalinear space partitioning.
- Force fill-space – to expand the existing Gems to bump up to each other and fill "non-gem" defined space.
- Flat top – to allow easy extension of Gem-space ceilings as required.

We will flag part of the Gems structure to indicate which planes are being used.

Scene Manager Data Organization

This section describes the primary data organization for the scene manager. There are 4 trees based upon the RTree type *(See Appendix A)*. The following block diagram shows the 4 data trees with sample data contained within them.



- Broken Lines show pointer connections between nodes that are not tree related. An arrowhead pointing to a node indicates that the node on the other end of the line points to it. Also note that some nodes point to each other, shown by an arrowhead on both sides of the line.
- The various cross-hatch is shown to indicate that there are 3 hierarchical objects in the template tree, one with 3 nodes, one with 2 nodes, and one with 1 node. Note the same patterns in the instance tree indicating that those nodes are instances of objects in the template tree.
- Connections not shown are that each individual node in the instance tree points to it's template node in the template tree.

Scene Manager Task Processing

The scene manager task processor contains a simple core (some call this the "main game loop"). The simplified task processor looks like the function below:

```
status DoScene()
{
   BOOL done = FALSE;

   while(!done)
   {
       UpdateTime();      // Update simulation timers and frame count.

       GetControllers(); // Get game controller info.

       done = DoAI();     // Call each controller performing AI.

       if(!done)
       {

             DoAnimation();     // Position each instance node based
                                // upon animation data in the template.

             PositionCamera(); // Update the current camera.

             DoRendering();     // Render the scene.
       }
   }

   return statusCode;     // No status processing in this example.
}
```

Note that DoAI is basically implemented as follows:

```
treeWalk(pControllerTree,callbackAI,NULL);       // (See appendix A)
```

Animation

The animation system is a key-frame based playback system used to animate various attributes of a hierarchical skeleton-based model. In addition to traditional game animated qualities, light parameters and events may also be animated.

Animation Data

There are 4 categories of animation data supported. These categories are:

- Geometry animation (including camera)
- Light attribute animation
- Object attribute animation
- Event spawning animation

Geometry animation

Geometry animation is used to translate, rotate, and scale an object relative to the object's local coordinate system (or pivot point animation). The method used is a stack-based system, where the first object in a hierarchy is brought into the world space coordinate system, then each object in the hierarchy is then brought into it's parent's (world modified) coordinate system in turn.

The geometry animation does not actually effect the vertex information of the model. The result of the geometry animation is saved in each object's "instance node" in matrix form. This allows the new world coordinate system matrix to be available for positioning of the object's collision data during collision checking, for AI processing, and for supplying to the renderer along with the vertex information.

Light attribute animation

Light attribute animation is used to animate properties of a light in the scene. While the actual "geometry" of the light is not shown, lights may animate freely in a game scene. Note that if you wanted to show the "source" of the light, the artist is free to build an object that looks like a light, (such as a car headlight) and hierarchically attach a light object to the headlight. Also note that Geometry animation is available for lights as well.

These animated values effect the attributes of the object's instance, and is made available to the renderer. There are 3 types of lights supported in Mud, Sweat & Gears:

- Omni lights, which give light in all directions.
- Directional lights, which give light in one direction as a plane (also called planer lights).
- Spot Lights, which cast light in a cone like a flashlight does.

There are six light attributes that will be animated. The first four apply to all lights, while the last two will only apply to the spot lights.

RGB color

Red, green, and blue may be animated independently from 0.0 to 1.0.

Ambient lighting (animation)

Ambient light properties of object may be animated. The rendering system will use this to change the ambient properties of the material.

Intensity multiplier

The intensity of the overall light may range from 0.0 (off) to any positive value. This effects the intensity of the light. This may be used not only to raise the light value, but may also be used to determine the brightness of lens glare and lens flare.

Start attenuation

As a light's range to objects increases, the effect of the light diminishes if the start attenuation of the light is less than the range of the light. The intensity between the start attenuation and the range of the light will fall off by a TBD function. Likely falloff candidates are RMS (root mean square), logarithmic, or table driven.

Range

The range of the light is the range where the light stops having an effect on the scene. This is a sharp cutoff. If a more gradual cutoff is desired, setting the start attenuation value will allow the light to appear to dim until the end range is reached.

Umbra & Penumbra

The umbra and penumbra are for spotlight parameters only.

The **umbra**, sometimes called the "hot spot", is the brightest part (at the center) of a spotlight. The radius of the umbra may be animated.

The **penumbra** is the darker "outside" area of a spotlight. The radius of the penumbra may be animated, which will increase or decrease the radius that the light covers.

Object attribute animation

Object attribute animation covers attributes that affect the look of the object in a non-geometric fashion. Currently, the only attribute that falls under this category is visibility. Once again, this attribute is done during the animation phase and is available not only for the renderer phase, but also during the AI phase, where it would be possible to have objects that are not seen by other AI objects... or objects that were hard to see (if such AI was required).

Visibility refers to the translucence of the geometry of the object, where 0.0 visibility is completely invisible, and 1.0 is fully visible. This is used to fade objects in or out when appropriate, such as an explosion that is constructed out of a hemisphere, where scale will expand it, and as it gets near it's maximum size animated visibility causes it to quickly fade away.

Event spawning animation

Events may be imbedded within an animation where the AI for the object under animation interprets on a case-by-case basis. Using the example of a tire with a nail in it, an animation could be used to spin the tire. When the animation got to the point where the tire would normally be in contact with the road, the artist could place a text event on the animation named "click". The AI controller that is in control of this animation would receive a text event at the proper time in the animation, where it could then check if the tire was in contact with the ground and play the "click" sound.

Adding sound events

Sound events are another form (although similar to the last example) of event spawning. In this the artist applies a sound event to the animation, such as a foot step noise in a run animation. Each time that the object's foot reaches the proper point in the animation a sound is played. This requires no further support by an AI programmer and is handled by the animation system.

AI/Controllers

AI for each object is handled in the AI phase of the game loop. The scene manager calls each registered object in turn, and the objects have an opportunity to do what ever action they deem appropriate. There are three types of object in the game that have complicated AI – cars, terrain, and camera. The camera is treated separately in the "Camera System" section.

Car AI

To first approximation, NPC drivers will attempt to follow a spline-based path around the track. There will be decision points embedded in the spline data that will enable the drivers to take alternate paths, if such exist and are "open." NPC cars follow pretty much the same physics as the player cars, so the NPC driver must make decisions about how much gas, brakes or steering to apply to get the car to follow the splines around the track.

To second-order approximation, the car AI will look for objects somewhat ahead or behind, and attempt to avoid or collide with them, as appropriate. Different NPC drivers will have different driving abilities, and will be more or less friendly toward the player depending on the NPC's goals.

Risks - Medium

The risk in car AI is processing time.

As currently envisioned, every NPC car will be interacting with the terrain at the same level as the player's car. This means the full physics and collision model is active for each NPC car, that terrain data must be available to every NPC car, and that terrain deformation must be calculated for every car. That's a lot of processing.

We do not currently know how long this processing takes, and we may find that it is not feasible to run this full model for cars that are off-screen. If we find that the processing takes too long, we will drop down to either a simplified physics model or a statistics based determination of events off screen.

Terrain & Deformation Controller

Terrain AI is what the terrain does in response to cars driving over or skidding across it, or to other events. Some percentage of the terrain is deformable, meaning that ruts will appear when cars drive over it. The physics engine will provide the terrain AI with information about how hard the tires are pressing into the ground, as well as the locations and size of the tires, and the terrain AI will decide whether to create ruts. Since we do not want cars to be able to create arbitrarily deep ruts as they cross the same terrain over and over again, we will maintain rut maps with information about how much the terrain has already been deformed. Past a certain deformation, ruts get no deeper.

- **Ruts** themselves are implemented as displacement maps, and either the terrain engine or the rendering engine will be responsible for subdividing the polygons in existing landscape to create the ruts.
- **Sandbars** are a special case of rut-able terrain. Instead of being deformable to a certain depth from the original surface, sandbars are deformable to a set y height. This will allow trucks to nearly totally destroy sandbars as they splash through water.
- **Splashes** in water are also part of the terrain AI. When a car drives through water, the terrain AI places a time-variant displacement map on the water (the big splash), and also calls to the particle system to create some water particles (the little splooshes).

In general, the terrain system will call to the particle system for particles when the physics system indicates that there is something skidding or sliding over the ground, and the terrain system determines that the terrain is of the correct type to create flying particles.

Risks - Medium

The risk in the terrain AI is in the number of additional polygons the displacement maps may create.

At the moment, we do not know exactly how many polygons we can process, or how many polygons we need to make our scenes look good. In the current design there aren't that many areas, but we may need to limit these still further.

A further discussion of AI and our approach to implementation

The AI for our game is broken down into a set of operations that are performed per time slice. Because the trucks in this game will be moving several inches per frame (at 60 fps), our time slices must be fractions of frames to avoid objects traveling through one another.

Optimally we would like to update our truck AI 8-10 times per frame. Of course not every object needs to be updated that often and some objects don't change at all. Luckily we have a system of controllers with priorities that allow us to decide how often each object AI should be updated.

As described elsewhere, objects are categorized by whether they can be animated or changed in response to events. In this case we differentiate objects by whether or not they are controlled by a physics system.

Whereas most objects will have a simplified control flow:

1. Change to next animation state
2. Move to new position based on animation path
3. Check for collisions that have occurred from movement
4. Generate events caused by collisions (such as change animation state or start a particle system)
5. Move objects to resolve collision
6. Loop

The control flow for a truck time slice is as follows:

1. Get input
2. Change animation based on input
3. Move to a new position based on the animation path and/or velocity previously calculated by the physics system
4. Check for collisions that have occurred from movement
5. Calculate forces (from collisions, input, etcetera)
6. Generate events caused by collisions
7. Move objects to resolve collision
8. Update object velocity
9. Loop

Master or Slave?

The question becomes, which comes first – the AI or the physics? Which is in control? Does the physics system move the truck, maintain all the object structures that contain position, orientation and status, update the object matrix and basically resolve all collisions? Or is it up to the AI to do so, and accept only suggestions from the physics systems as to where objects should be moved, making the ultimate decision. Let's dig in...

Even within each approach there are multiple ways for the control to go. So we have analyzed two of them here.

AI is the master, physics is the slave

(physics gives suggestions to AI)

In this scenario, each controller is processed in turn.

The control flow for a truck time slice becomes as follows:

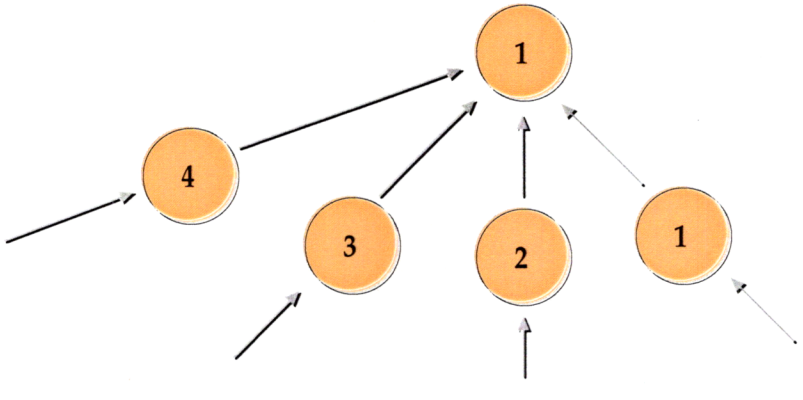
1. Get input
2. Step through each controller, getting each object instance
3. For each controller,
   - i) AI changes object's animation based on input
   - ii) AI moves object to new position based on animation path and/or velocity previously calculated by physics system
   - iii) call physics system to check for collisions that have occurred from object's movement
   - iv) physics system calculates forces generated from collisions and latest input
   - v) physics system sends messages to each object affected with collision event, and collision data: forces, velocity, orientation, etc
   - vi) AI checks message queue for current object to see result of collision.
   - vii) if there is a collision, AI moves the object to resolve the collision using the physics data and updates any damage system or animation state, etc. The AI may choose to loop back to iii) a set number of times to make sure that where it is placing the object does not cause another set of collisions.
   - viii) AI updates the object velocity based on physics data for next time slice
4. Loop

Let's imagine a worst case collision scenario:

Four objects are about to collide simultaneously. What happens?

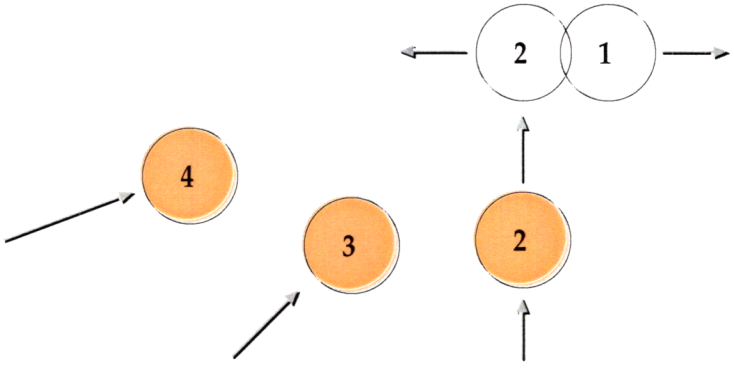
Controller 1

Object 1 moves into its new position. It's all by itself. No collision. No problem.



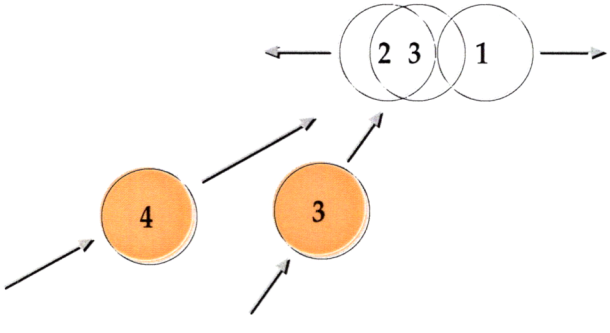
Controller 2

Object 2 moves into its new position and collides with object 1. The physics system calculates the forces and send the events and collision data to the message queues of object 1 and object 2. Object 2 immediately processes its collision data and moves safely out of the way. Object 1 waits to be processed next time around.



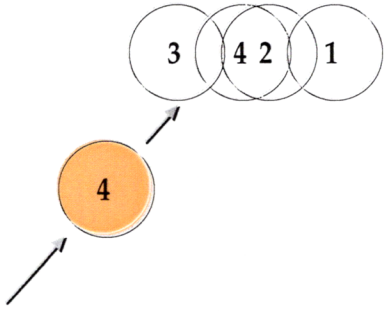
Controller 3

Object 3 moves into its new position. Oops. It's now occupying the same space as Object 2 *and* its colliding with Object 1. Okay, no problem. Physics system calculates the forces and send the events and collision data to the message queues of objects 1, 2 and 3. Object 3 immediately processes its collision data and moves safely out of the way. Objects 1 and 2 patiently wait.



Controller 4

Object 4 moves into its new position. Oops. It's now occupying the same space as Object 2 *and* its colliding with both Objects 1 and 3. We know the drill. Object 4 is moved and messages are waiting in the queues of objects 1, 2 and 3.



Message Queues

So let's examine the message queue for each object as the collisions progress:

MQ4	MQ3	MQ2	MQ1	
			No collision	Object 1 moves
		Collision *Collision resolved*	Collision	Object 2 moves
	Collision *Collision resolved*	Collision	Collision	Object 3 moves
Collision *Collision resolved*	Collision	Collision	Collision	Object 4 moves

When it becomes time to resolve Object 1's collision data, it will have compounded multiple forces that propel it way beyond its expected range.

Physics is the master, AI is the slave.

In this scenario, all objects are moved before collision detection occurs resulting in equal forces being generated as a result of multiple object collisions.

The control flow for a truck time slice becomes as follows:

1. Get input
2. Step through each controller, getting each object instance
3. For each controller,
   i) call AI to change object's animation based on input
   ii) physics system moves object to new position based on animation path and/or velocity previously calculated by physics system
4. Step through each controller, getting each object instance
5. For each controller,
   i) physics system checks for collisions that have occurred from object's movement
   ii) physics system calculates forces generated from collisions and latest input
   iii) call AI for each object affected with collision event to change graphics, animation state, update damage system, etc
   iv) physics system moves each object to resolve the collision using the calculated physics data. The physics system may choose to loop back to 4) a set number of times to make sure that where it is placing the object does not cause another set of collisions.
   v) physics system updates the object velocity for next time slice
6. Loop

The major drawback with this approach is almost immediately obvious — the controller hierarchy is traversed multiple times.

Best of both worlds: The solution

In this scenario, we split the duties of both AI and physics to run in logical sets.

Move activity occurs first, flagging collisions and sending to a new Collision Candidate List (CCL) for later processing. All movement activity happens in this once set.

Following this, we walk the CCL to calculate forces, sending all this information back to the pertinent controllers in the source tree. This process continues for calculating all forces caused by interpenetration.

The last step is to call events for collisions, move to resolve collisions and update velocities, then recurse *x* number of times to the top. This is the critical move that allows us to continue collision chain events, but before it becomes a problem computationally *(in processing time)*, we quit – flag the tree walker to stop the insanity and switch to a priority based shuffle-board mode – to space out the remaining objects and get on with the business of the game. At this point, we've made our due dilligence to have accurate appearing physics.

1. Get input
2. Step through each controller, getting each object instance
3. AI – part 1: For each controller
   i) change object's animation based on input
   ii) move object to a new position based on animation path and/or velocity previously calculated by physics system.
   iii) call physics system to check for collisions that have occurred – flag collisions that have occurred, adding to a Collision Candidate List
4. Physics
   i) walk the Collision Candidate List to calculate forces generated by collisions and input.
   ii) send all data regarding forces, velocity, orientation, etcetera - back to the a affected objects in the source list.
5. AI – part 2: Clean-up
   i) call events for collisions (sounds, damage system updates, etcetera)
   ii) move to resolve collisions
   iii) update velocity based on physics data for next time slice.
6. Loop *x* times to 4 or break out.

Quick note: An object can have a flag set in it's properties that it should not be handled in the default manner. For instance, it could say that it is not subject to recursion of the collision system, or that it has priority for one reason or another.

Support AI types

Certain AI objects are used to support the engine and the look of MSG.

Triggers

Triggers are another AI type that may be given to objects placed in the scene with Calliope™. Triggers are special "event generator" objects that allow the artist to control various stages of a level using timers or milestone areas in the scene to send messages to the engine's controller tree. These messages are then intercepted by other AI objects waiting for them, such as FMA director controllers, a particle emitter that emits leaves blowing as the vehicle speeds by, or a herd of deer on the side of the road that scatter as the player's vehicle roars toward them.

There are 3 basic types of triggers, which allow a wide range of control during scene game-play layout. Each type of trigger sends a text message. How these messages are interpreted is up to the AI of each controller. Note that any trigger may have the option to remain in the scene and re-arm itself after a period of time, or to remove itself from the scene after firing. The following is a description of the triggers:

Event triggers

Event triggers fire at specific events during the game. For the two types of events that require triggering at the moment, laying actual triggers is not required. The engine sends a trigger message titled "scene start" when the scene starts, and "scene end" when the scene is ready to end. These are required right now to allow FMA sequences a chance to start, and have no other known purpose at the present.

Timed triggers

Timed triggers maintain count-down timers that will fire when a certain time is reached. These could be used to denote that an event such as an earthquake or a flood should start now (assuming there is an AI built to receive these messages).

Timed triggers may be set to fire once, or to repeat at an interval. Both the first firing and subsequent firing (if set to repeat), have their own time values that may be set, as well as a min/max range to optionally support random intervals between timed events.

Collision triggers

If a trigger has a collision node hierarchically attached to it, then the trigger may fire when a collision occurs. This trigger has the option of defining categories of objects that when collided with it will cause the trigger to occur. The following is a list of the categories that may set a trigger off when colliding with the trigger object:

- Any object
- Player's vehicle
- The exact name of another controller

Swappable objects

Swappable objects are a controller type that are used to swap one object with another. Their Calliope™ parameter is the trigger name that causes the swap.

It is up to the artist to then link the starting object as a child to of the swappable object controller, and then to link the "swap to" object as a child of the starting object. When the trigger to swap the objects occurs, the starting object is deleted, and the "swap to" object is replaces it and starts up it's AI.

By placing a collision trigger object at the same location of the swappable object, and having it trigger the swap, the engine gains the ability to have a vehicle appear to collide with the object and have the object change it's behavior and/or appearance, such as a haystack that when hit seems to explode into fragments of hay (due to a particle generator being the "swap to" object that emits hay pieces).

Scenery generators

Scenery generators are a form of geometry compression. They allow us to have a much higher volume of geometry in the world by a form of "meta instancing" objects. The are used to populate forests with trees, where the actual geometry count is taken up by templates of only a few trees, which may or may not appear in the scene in other places as their own objects.

Note that trees are only an example application, as the other attributes described below make it possible to populate areas with any type and combination of objects.

Scenery generators have only a volume for geometry, and the volume is inhibited from rendering. This may be as simple as a surface where objects are instantiated on, which can have a polygon count as low as 1 triangle, or may have 20 or 30 (or more) that define a surface over a hill or in a glade.

Scenery generators also have a target count of objects to place randomly over the surface, and a list of objects to choose from to populate the surface. Each object in the list has a target percentage to stay close to during the population. Each object also as x/y/z scale ranges which can be used to define differences in the list of objects as they are placed.

The last attribute the scenery generator has is a seed value to use when plotting the distribution of these objects. When the region containing the generator is visible, the generator will create from the seed a distribution of the objects in the list over the given surface. Using the seeded random generator, each object is scaled within the range given by it's x/y/z scale values, and is rotated horizontally a random amount to create diversity in the area. The method of generation will produce identical results each time it is built, so the landscape may become familiar to the observer.

This same generator will be built into the Calliope™ plug-in so that the artist may see the results in 3DS MAX for a given area, and play with the given attributes until satisfied with the result.

Rendering

The renderer is divided into two pieces - a **high level renderer** and a **low level renderer.** Press Start is responsible for the high level renderer and for defining the interface to the low level renderer. Interplay is responsible for creating the low-level renderer.

Each frame, the Scene Manager goes through the list of regions and decides which are visible, based upon the visibility list from the region that the camera(s) is(are) in. It takes the list of objects in these regions, does some additional high level visibility testing, and passes a list of visible objects down to the high level renderer.

The high level renderer decides what LOD the objects should be displayed at, sorts the objects based upon priority and transparency, and, come rendering time, sends down a list of polygons to the low level renderer for displaying.

The high level renderer also maintains knowledge of the low-level renderer's state - lights, viewports, special PS2 effects, etc - and sends state change information to the low level renderer as necessary.

The Scene Manager specifies which lights and cameras are in the scene, but it is left to the high level renderer to package up any state change information and send it to the low level renderer.

The low level renderer is responsible for transforming, lighting, drawing and texturing the triangles. It is expected to be able to handle single triangles, triangle strips, triangle fans, points, and lines. It is expected to be able to handle transparency, translucency, backface culling, fog, multiple textures, texture blending functions (such as the PS2 supports), and reflections.

Requirements for a Renderer

Our scene description is a hierarchical list of all the objects in a game level organized by location or region. Each frame, the Scene Manager traverses this list handing off objects to the renderer to be displayed on the screen.

Many factors affect how each object appears:

- Time of day
- Weather
- Damage
- Mud
- Water
- Distance from the camera

These can drastically change the physical appearance of the object, and the renderer must take into account each of these factors.

Object Description

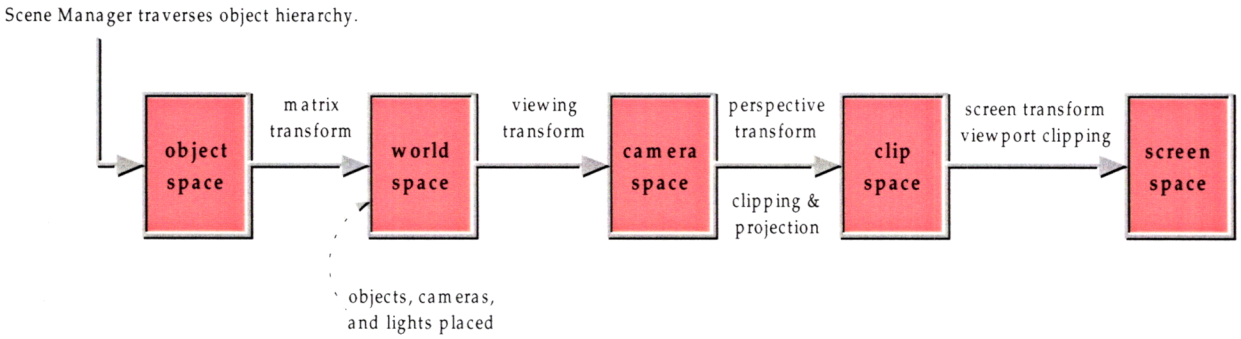
Each object is broken down into each of its moveable or detachable parts. These subobjects can have their own texture, color, material properties (such as how reflective or mat the surface is) and can move independently. Typically the object will be decomposed into triangles, although we are looking into possible curved surface descriptions.

A vertex or control point has the following information associated with it: coordinate data, x, y, z; vertex normal to indicate curvature, nx, ny, nz; texture data, u, v, per texture (there may be several textures).

The Conceptual 3D pipeline

The conceptual 3D pipeline takes objects which have been defined in a local coordinate system called *object space* and assembles them into a scene or *world space* according to *object matrices* which specifies their position and orientation in the world. Subobjects are assembled into objects and objects into scenes each with their own set of matrices.

Once the world has been assembled, lights, such as the sun, are placed in the world with a position and orientation. Lights, such as headlights, can also be attached to objects and can move with the object. Finally, cameras are placed in the world to determine how much of the world you'll actually be seeing and from what viewpoint. One you have chosen a camera, a *viewing transform* is applied to the objects to place them in the chosen *camera space*. The contribution of all light sources is calculated, texture is added, perspective is applied and shadows are projected. Object triangles are clipped to a viewing frustum, a truncated pyramid, and then projected onto a 2D plane. The remaining visible triangles are converted to *screen space* and clipped to either the screen boundaries or defined viewports which are subregions of the screen.



Graphics Pipeline

In reality many of these transformations can be combined. Typically the object, viewing and projection matrices are concatenated together into one matrix and vertex coordinates are transformed once.

Lighting can be performed in object space by transforming the light vectors into object space or by transforming the normals into camera space.

The GS only accelerates the final stage of the pipeline. Our renderer must be able to take objects in object space, process the entire pipeline before handing triangles off to the GS.

Lights, Shadows and MultiPass Rendering

There will be many types of light sources and lighting effects in Mud, Sweat and Gears. For dynamic lights, we will need to be able to change a light's position and direction, turn it on and off, and set its physical characteristics such as the color of the light, its intensity, whether it's a spot light, how wide an angle it covers, etc.

Static light contribution will be precomputed in a light map. Light maps are low resolution textures that are indexed into like any other texture and combined with the results of previous lighting and texturing operations. This use of multiple textures implies multi-pass texturing through the GS, storing intermediate results, and applying the various combine modes with each subsequent pass.

Shadows are dynamic and static according to their light source. Again, static shadows will be pre-computed and either stored as textures or as polygons and combined with previous operations.

Dynamic shadows require more computation. A *shadow map* can be created by rendering the scene or portion of the scene from the point of view of the light source. Then the same scene is rendered from the

point of view of the camera (as it normally is). If an object is in front of the stored shadow map value, it is lit, if it's behind, the object is in shadow.

Here are some of the existing structures used to define lights:

```
typedef struct
{
    psRGB      color;
    float      brightness;      // MAX multiplier setting.
    float      attenStart;      // Range that light intensity falloff starts.
    float      attenEnd;        // Maximum range of light.
    float      hotSpot;         // Radius of innner cone.
    float      falloff;         // Radius of outer cone.

} Light;

typedef struct
{
    LightType  type;
    float      posX, posY, posZ, posW;
    float      dirX, dirY, dirZ;
    float      ambR, ambG, ambB;
    float      diffR, diffG, diffB;
    float      specR, specG, specB;
    float      spotExponent;
    float      spotCutOffAngle;
    float      attenuation1, attenuation2, attenuation3;
} LightDesc;

typedef struct
{
    float      ambR, ambG, ambB;
    float      diffR, diffG, diffB;
    float      specR, specG, specB;
    float      specExponent;
    float      emissR, emissG, emissB;
    int        numTextures;
    int*       pTextures;
} MaterialDesc;
```

More on lighting

Verticies of an object are lit using the following set of equations.

color = *ambmat* * *ambenv* + Σ (att)(spot)[(*ambmat* * *amblight*)
+ (*diffmat* * *difflight*) (ndotl)
+ (*specmat* * *speclight*) (ndoth)^(*specexp*)]

where:

ambma = ambient material property

diffmat = diffuse material property

specmat = specular material property

amblight = ambient light component

difflight = diffuse light component

speclight = specular light component

specexp = specular highlight of material

Each light's contribution on an object is computed individually and summed together to determine the global light contribution. The initial ambient component is the ambient contribution of the environment: a sunny hillside will have a higher ambient value than a dark forest. The second ambient component is determined by the material properties of the object and how it responds to a light source.

The diffuse component is determined by the material properties of the object and the diffuse component of a light modified by the dot product of the normal of the object and the light direction. Remember that lights are positioned in world coordinates, but objects are defined in local modeling coordinates. In order to do this calculation both the lights and the normals must be in the same coordinate space. Either the normals must be transformed into world space (or camera space for reasons below) or the lights must be transformed into the object's modeling space. In the first case, the normals are transformed by the transpose of the inverse of the matrix used to transformed vertices into world space (or camera space). In the second case, the lights must be transformed by the inverse of the same matrix back into model space.

The specular component is only computed if the dot product of the light and object normal is greater than 0, in other words, only if the object is being illuminated by that light source. First the half angle between the camera and the light is computed and raised to the power of the specular exponent. The half angle is where the specular highlight is most intense. The larger the specular exponent, the brighter and narrower the highlight. A less intense highlight is spread over a larger area of the object. Since the camera direction vector must be in the same coordinate space as the light vector, one or the other must be transformed into the other's coordinate space. The specular component is finally determined by the material properties of the object and the specular component of a light modified by the dot product of the normal of the object and the previously computed half angle.

Attenuation is computed as a function of the object's distance from the light source.

$$att = 1/[att0 + att1(dist) + att2(dist)^{(2)}]$$

where:

att0, att1, att2 = light attributes

A spot light's contribution also takes into account the shape of the spot light: the umbra is the central cone of the most brightness. The penumbra is the outer cone of lesser intensity and potential falloff.

$$spot = ([\rho - \cos(\theta/2)] / [\cos(\theta/2) - \cos(\phi/2)])^{(falloff)}$$

where:

ρ = (ndotl)

θ = umbra angle of spotlight in radians

ϕ = penumbra angle of spotlight in radians

Reflection Vectors, Environment Mapping

Reflections and particularly *environment mapping* enhances the realism of shiny objects. A reflection vector is computed that is normal to the surface and used as an index into a pre-computed, coarse texture map.

Transparency and Translucency Means Sorting

Both textures and object material properties can have alpha values that are less than 1.0 indicating transparency or translucency. Because of the nature of Z-buffering, triangles that contain alpha values less than 1.0 must be rendered after all other triangles are rendered and then sorted so that they are rendered back to front. This must be done by the renderer, as the Scene Manager only knows about subobjects, and not about individual triangles.

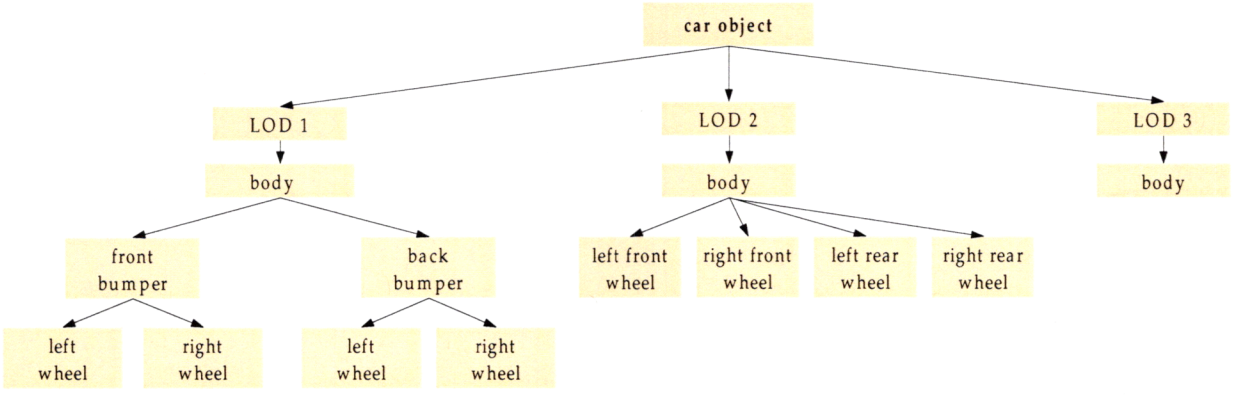
Levels of Detail (LODs), Bounding Calculations and Interpolation

One way to reduce the amount of data rendered is to create *levels of detail*. Since objects far from the camera are smaller and appear less detailed, it is reasonable to portray them with less detail than when they are close to the camera. There are several approaches one can take to doing this. The most common way is through mipmapping, which reduces the level of detail in textures by storing prefiltered versions of the texture. Mipmapping is good for long walls into the distance which transition across many levels of detail. However, the number of triangles remains constant and some texture details which you'd want to be preserved are filtered away, such as lines for crosswalks or other distinguishing features.

Geometry LODs can reduce the number of visible triangles for distant objects, but still preserve features. Mipmapping typically can not be used in geometry LODs because feature reduction is selective and vertices from one LOD to the next no longer match.

Which LOD chosen is decided by either the distance from the camera or size of the object when its rendered on the screen. There is no point in rendering a 5000 or even 50 polygon object when it is only a few pixels on the screen. This presents a problem in our graphics pipeline described above. Distance from camera isn't known until vertices are transformed into camera space and size isn't known until the perspective transform is applied. It would be very time consuming to transform the vertex list of each LOD until the correct size or distance is found. Instead the scene manager will send the object bounding box or sphere down the pipeline, determine the correct LOD, and then send the correct object description.

One of the issues of using LODs is the popping caused from substituting one level of detail for another. We plan to avoid this by interpolating the vertices from one LOD to the next using a simple blend function.



Subobjects within an object hierarchy. Note each level of detail of an object may have a different number of subobjects.

Progressive Meshes Reduce LOD Data

Progressive meshes are a technique for reducing the amount of LOD data required using precalculated vertex splits to gain additional detail. All that is necessary is to store a coarse mesh and the sequence of splits. The renderer will need to traverse the split information adding vertices until the correct level of detail is achieved.

The one drawback of progressive meshes is that it requires all the data be in memory at the same time. With discrete LODs, only the 2 transitioning LODs need be brought into memory. We will be using a combination of both techniques, most likely progressive meshes for the terrain and geometry LODs for the cars and other objects.

Collisions Need to Be Computed in World Space

Associated with each object will be a collision tree of bounding spheres or collections of bounding spheres. The collision tree may not necessarily reflect the same hierarchy as the hierarchy of the object parts. Whereas the object part division is determined by how the object will be rendered or animated or disassembled, the collision tree division is determined by AI events.

Collisions must be computed in world space. This causes another break in our graphics pipeline. Most of the time a bounding sphere check will be sufficient. However, for those objects where precision is needed, such as the terrain and a car, individual triangles need to be checked. Therefore, an interim step of transformation to world space is needed.

Z Problems and Solutions

When 2 or more triangles have the same Z value, the order of rendering determines which one goes on top. If the order is not maintained from one frame to the next, flickering or sparkling occurs as part of the object disappears and then reappears. This also can occur because of precision errors from roundoff or truncation.

Z values that are very close may resolve to the same value. In particular when rendering objects on a terrain, its very important that the objects appear in front of the terrain and not falling through. One way to get around this is to always render the terrain first. Another way is to introduce *z-bias*, which pushes an object forward a set amount.

Some objects, like the sky, always need to be rendered first and some objects, like menus and overlays, always need to be rendered last. We will be using *z priority* to determine rendering order for these objects. Objects with a higher z priority will be rendered after objects with lower priority.

Surface Deformations and Displacement Maps

In Mud, Sweat and Gears, surface deformations, like ruts in the ground, waves in water, and mud troughs, will be created by displacement maps. A higher LOD mesh will be inserted around the collision area. Then each vertex of the mesh will be displaced according to 1 of 2 ways:

- For ruts, the displacement depth and length will be determined by the physics model.
- For mud and water, the displacements will be stored in a texture map.

Indices into the displacement map will determine the height of the resulting vertex.

Fog and Clouds

Fog and clouds will be pre-computed, coarse textures of density information. To give the illusion of depth, there will be layers of textures for given distances. Determining the fog value is simply a trilinear interpolation of textures between 2 different layers.

We will implement distance hazing by using the fog features of the Low Level Renderer.

Reflections

Reflections are handled by pre-defined environment maps. Reflective objects have a flag ("I'm reflective!") and an associated environment map. The low level render is expected to be able to do the multi-pass rendering necessary to make the reflection work. The environment maps themselves are low resolution textures drawn by the artists, and are not created dynamically.

Making Water Look Alive

Water moves in waves usually, so we will generate a function simulating waves to generate indices into the water displacement map. The new triangles generated from the displacements will also generate new normals to the surface. This will give us the appropriate reflections that you see in moving water.

Water flooding into a canyon will be modeled by successive geometry LODs that have been pre-fitted to the canyon walls. As water is transparent, it will have a high z priority thereby being among those objects rendered last. We will not be modeling defraction of objects submerged in water.

High Level Renderer API

light functionality

```
int     AddLight    ( Light* pLight );          // return lightID
BOOL    TurnOnLight ( int lightID );
BOOL    TurnOffLight( int lightID );
BOOL    MoveLight   ( int lightID, float x, float y, float z );
Light*  GetLight    ( int lightID );
BOOL    ChangeLight ( int lightID, Light* pLight );
BOOL    RemoveLight ( int lightID );
```

camera functionality

```
int     AddCamera       ( Camera* pCamera );  // return cameraID
BOOL    IsCurrentCamera ( int cameraID );
BOOL    MoveCamera      ( int cameraID, float x, float y, float z );
BOOL    RotateCamera    ( int cameraID, float x, float y, float z );
Camera* GetCamera       ( int cameraID );
BOOL    ChangeCamera    ( int cameraID, Camera* pCamera );
BOOL    RemoveCamera    ( int cameraID );
```

viewport functionality

```
int     AddViewport     ( int x, int y, int width, int height );  // return viewportID
BOOL    RemoveViewport  ( int viewportID );
BOOL    SetCurrentCamera( int viewportID, int cameraID );
BOOL    SetViewport     ( int viewportID, int x, int y, int width, int height );
```

scene functionality

```
BOOL    StartScene  ( InitParams* params );
void    EndScene    ( void );
void    SortScene   ( void );
void    RenderScene ( void );
```

Shadows

Shadows can be produced in a variety of ways. They come with different caveats. Render time and memory limitations will determine which shadow methods we choose.

Traditionally, shadows are produced by attaching a transparent sprite to the bottom of an object causing the shadow to move with the object. The shadow is then blended with the ground, indicating how the object is associated with the ground. These attached shadows do not shadow any object other than the ground. And since the shadow is fixed, the effect is as if the light is either directly overhead or firmly attached to the object.

To get a better approximation of shadows associated with the lighting of a scene, there is the concept of a projected shadow. These are typically done with projected textures.

See: Segal, Korobkin, van Widenfelt, Foran, and Haeberli, "Fast Shadows and Lighting Effects Using Texture Mapping", SIGGRAPH '92 for a more detailed description.

Typically a texture is pre-generated that is the shape of the object that is casting the shadow. These textures need only be 1 bit, transparent bitmaps and can be a coarse approximation of the shape of the object. During the game, a texture matrix is created that takes the dot product of the plane of projection *(such as the ground or a wall)* and the direction of the light, and uses that matrix to basically warp the texture image onto that plane based on the distance of the light from the object.

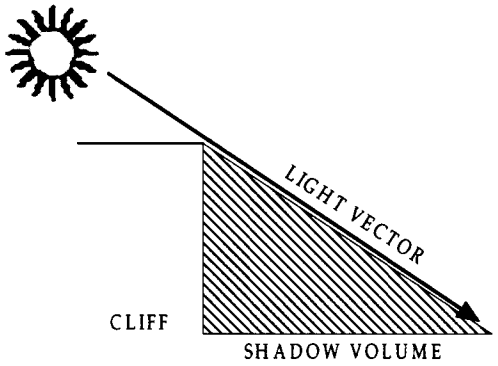
```
 dotProduct = projplane[X] * lightdir[X] +  projplane [Y] * lightdir[Y] +  projplane [Z] *
lightdir[Z] + projplane [W];

 texMatrix[0][0] = dotProduct - lightpos [X] * projplane [X];
 texMatrix[1][0] = - lightpos [X] * projplane [Y];
 texMatrix[2][0] = - lightpos [X] * projplane [Z];
 texMatrix[3][0] = - lightpos [X] * projplane [W];

 texMatrix[0][1] = - lightpos [Y] * projplane [X];
 texMatrix[1][1] = dotProduct - lightpos [Y] * projplane [Y];
 texMatrix[2][1] = - lightpos [Y] * projplane [Z];
 texMatrix[3][1] = - lightpos [Y] * projplane [W];

 texMatrix[0][2] = - lightpos [Z] * projplane [X];
 texMatrix[1][2] = - lightpos [Z] * projplane [Y];
 texMatrix[2][2] = dotProduct - lightpos [Z] * projplane [Z];
 texMatrix[3][2] = - lightpos [Z] * projplane [W];

 texMatrix[0][3] = - lightpos [W] * projplane [X];
 texMatrix[1][3] = - lightpos [W] * projplane [Y];
 texMatrix[2][3] = - lightpos [W] * projplane [Z];
 texMatrix[3][3] = dotProduct - lightpos [W] * projplane [W];
```

Again, projected shadows do not typically shadow any object other than the plane of projection. To shadow other objects, you would need to create a new texture matrix for each object or polygon within that object by inserting a new projection plane into the equation. On the other hand, if you're not overly concerned with exactness, you could use the same texture matrix for objects resting on the projection plane. The shadow could be grossly approximated by determining if the object was within the bounding area of the shadow and projecting the object's texture vertices onto the plane. This approach works best with static lights and static shadows. When the object that is casting the shadow is moving, or the light is moving so that the shape of the shadow changes, there is a problem. What people typically do in this case is create the shadow on the fly using the stencil buffer using multiple-pass rendering.

We are considering a modification of this algorithm recognizing that texture memory is limited. Instead of using a pre-computed texture, we could use a planar polygonalization of the image to create the shadow and the texture matrix to map the shadow to the projection plane. However, this method is, as of yet, untested.

Shadow Volumes

Shadow volumes are used to shadow more objects than just the ground plane. In this scenario, a volume is created using the silhouette of the object as projected on the ground plane (using a similar technique to the above-described projected shadows), the direction of the light vector, and the object, itself. When another object intersects this volume, the portion of the object that is interior to the volume (using the dot product of the light direction and the polygon normal) is considered to be in shadow. The shadow can be "applied" by darkening those vertices, or the lighting calculations can be skipped for those vertices effectively resulting in a shadow. Although fairly accurate, this approach is computationally more expensive than the previous approaches.



Dynamic shadows

For dynamic shadows, shadow maps provide more accuracy and the ability for soft shadows and shadow dropoff. The drawback is that this is another multi-pass rendering scheme and specific to a single light source. If you have multiple dynamic lights, you would need to create a shadow map for each light individually and then combine the results. In this scenario, objects within a scene are rendered first from the point of view of the light source. The z values are stored in a depth-map, or shadow map, which is treated like any other texture only it contains floating point values. Then the scene is rendered as it normally would from the point of view of the camera. The world coordinates of each polygon are transformed back into light coordinates and tested against the z-value stored in the shadow map. If the coordinate is behind the stored value, the object is in shadow. Otherwise the object is lit. The further back the object is from the light source, the softer the shadow. Obviously, this approach has the drawback of requiring more memory for each shadow map than the previous approaches and takes longer time to render due to multi-pass rendering. However, since we are only interested in the z information the first time through, it is a somewhat faster rendering stage —texturing, lighting, color blending is ignored.

Risks - Medium

There are few risks with the high level renderer. It is simply an interface with between the Scene Manager (and possibly other modules, if we find they need to call rendering functions directly) and the low level renderer. However, the low level renderer has its risks.

The low level renderer will be written entirely in VU1 assembly, and will be very difficult to debug or change. Further, the number of polygons we can put in the game may be limited by the speed of the high level renderer. Since we do not yet have a PlayStation 2, and have not yet gotten a feel for where the bottlenecks in our renderer are, we may end up making the wrong speed trade offs in the low level renderer. These may be difficult to correct later on.

Camera System

Providing a method of decoupling the camera has been a classic problem for 3D games since the beginning. Many games use a direct method that can give you a headache (like DOOM-64) or they put the vehicle on a Popsicle stick, like a lot of Indy and Formula-1 racing games we've all seen. It gives you the feeling that the vehicle isn't attached to the road, but rather to the screen – and for good reason, it is.

This Camera System decouples the active camera from the action by two degrees, solving the traditional Popsicle problem (and others as well).

It allows you to set a target for the camera to track *(internally called the Track Target)*, by setting and managing four values:

- The **track target** x, y, z (target object's position) updated each frame (if the target is moving).
- The **distance** to keep from the target object.
- The **locale** (or angle) about the target object on the X/Z plane.
- The **altitude** to seek relative to the target object.

The active camera seeks its Surveillance Point using the D-O algorithm.

The code facilitates multiple cameras and could be easily extended to have cameras automatically follow Splined paths while switching which camera is active. An engineer could, for instance, put several cameras into motion about an object and randomly switch between active cameras.

The Surveillance Point (or SP) is key in the success of this system, because it is where the camera is **seeking to be**. The SP in turn, seeks it's place relative to the Track Target (or target object). Both use the same method, the simple D-O: **(destination – origin) / aggression**

Depending on the aggressiveness of the camera object, it will move more or less quickly to its target location. This type of camera is ideal for following moving objects, since its motions will complement the motions of the objects themselves. There are two types of aggression:

- **Pursuit Aggression** – internally referred to as just "aggression." It manages how quickly the camera reaches its Surveillance Point (SP). This is set using the function SetAggressive(x).
- **Locale Aggression** – manages the aggressiveness of the Surveillance Point reaching its Locale (angle) about the track target on the X/Z plane. This is set using SetLocaleAggressive(x).

Setting Locale about an object will *short circuit* – meaning that if you set it to 10 degrees, and then set it to 350 degrees, it will pass through zero to get there instead of making the longer (and senseless) journey.

Managing the Camera System is very easy with the primary controls, however there are times when you might want to bypass the primary operational method. In this circumstance, one might be tempted to manage the Camera directly. ***Moving the camera directly is a bad idea*** – so in the spirit of decoupling our camera from the target object, an alternate service is provided:

Once set into the alternate pursuit mode, you manage tracking of the target by using SetViewReference() and manage the camera's position by using SetSurveillancePoint().

Risks - Low

The code already exists and has been used in other games. The only real issue is that the code as written does not deal with objects occluding the view of the target object. A higher level AI must be written to recognize this case and either move the camera or turn off the visibility of the offending object.

Specification

Definitions:

View Point *(also known as VP)*

Where you are... the camera's current position.

View Reference *(also known as VR)*

What you are looking at, or tracking.

Surveillance Point *(also known as SP)*

The point where you want the camera to go, the SP maintains it's position relative to the target.

Locale

An angle about your target in degrees, the locale operates in the XZ plane.

Aggression Level *(Pursuit Aggression, Locale Aggression)*

Defines how we pursue the target, ranges LAZY to STAPLED-ON

Track Altitude

How high (or low) we put the camera relative to the target.

Track Distance

How far we must stay away from the target on the X/Z plane.

General CS Functions:

InitCamera(*camera, distance, delta_height, locale_angle, tx, ty, tz)

This function creates your camera position by knowing the target object's position *(tx, ty, tz)* and your angle in relation to it (locale angle). To do a "fly-in," use this function and then force the camera to a temporary position using PutCamera(). Setting the angle here will **snap** to that Locale, but don't use this function for that, use ResetLocale() instead.

Parameters required for this function (in order) are:

- Pointer to an instance of a CAMERA object
- Distance camera should keep from target
- Height of camera above target (relative height, could be positive or negative)
- Locale angle about target (in degrees)
- Target coordinates x, y, z

Passing a NULL pointer to this function in lieu of a CAMERA object will result in using the default camera. Each time a CAMERA object is initialized with this function is becomes the current camera as well. To specifically choose a camera, use the function SetWhichCamera(*camera);

SetTrackTarget(x, y, z)

Tell the camera system where to look – *If your Target is moving, make successive calls to this function.*

UpdateCamera()

This is the camera tracking update function. It should be registered with the Scene Manager or V-Blank Interrupt to be called once per frame – this guarantees frame-rate independent camera motion. There are two operative modes used in the normal operation of the Camera System. They are controlled by using the *SetCameraMode()* function:

```
SetCameraMode(IDLE);
```

or `SetCameraMode(PURSUIT_MODE);` (default)

PutCamera(vpx, vpy, vpz)

This function forces the actual position of the current camera to the specified coordinates, it is a temporary setting because the primary behavior of the Camera System is to seek the Surveillance Point.

The camera will smoothly relocate from its current location to the Surveillance Point as UpdateCamera() is called *(unless SetCameraLock(ON) is invoked).* The Surveillance Point is set initially in InitCamera() and is updated in real-time while the target is moving *(every frame UpdateCamera() is run).*

SetLocale(angle_degrees)

Rotates Surveillance Point about the track target. This is the primary means of controlling where the camera is located on the X/Z plane about the Target. Update this value to quickly swing the camera from the front of the car to the back, or the sides *(presuming of course, that you are using the vehicle as the target object).* It could also provide an interesting visual for sliding around corners if one were to link the car physics to this value.

GetLocale()

Returns the Locale setting for the current camera. This is the actual locale setting, not where it's intending to go. See GetLocaleDest() to learn where the camera is currently trying to go.

GetLocaleDest()

Returns destination angle for Locale of the current camera (not it's current Locale).

ResetLocale(angle, locale_aggression)

Snaps to a Surveillance Point about the current Track Target. Locale aggression levels are 0-8, refer to the SetLocaleAggressive(*x*) function for more information.

SetTrackAltitude(delta_height)

Sets the camera tracking altitude above tracked object for the current camera. User specified, no default.

GetTrackAltitude()

Returns the tracking altitude of the current camera.

SetAggressive(value) *(Pursuit Aggression)*

Sets the pursuit aggression value for the current camera. The "aggressive" is set in the range of 0 and 6, with zero being disgustingly aggressive. This is what other *primitive* camera systems would use, giving you the impression that the vehicle is on a Popsicle stick; might as well not use the Camera System then.

A value of 2 or 3 is a normal value for this type of aggression. A value of 6 is very slow in catching up.

Can be set in the range of 0-6; 0=most aggressive, 6=least. **Default is 3.**

GetAggressive()

Retrieves the current camera's pursuit aggression value.

SetLocaleAggressive(locale_aggression) *(Rotation Aggression)*

Sets the Locale aggression value for the current camera. The "aggressive" is set in the range of 0 and 8, with 0 being the equivalent of NO tracking at all. Basically it puts the camera at the Locale immediately. Using a value of 2, 3, or 4 is normal for most operations. Eight is the going to make it crawl like a snail.

Locale aggression levels are 0-8: 0=stapled on, 8=slow rotation. **Default is 3.**

SetTrackDistance(distance)

Sets the distance of camera from the tracked object. User specified, no default.

GetTrackDistance()

Returns the tracking distance for the current camera.

SetSourceOverride(ON|OFF)

This command is not technically part of the Camera System, as it simply sets a flag that the game engine uses. It's purpose is to ignore settings from the AI; we leave it up to the AI to pay attention to this flag.

For instance, you would use this if you wished to temporarily take control away from the normal AI and say... perform an Instant Replay or cinematic sequence – or some other purpose. The function is benign as it only sets a value in the active camera's structure. **Default is OFF.**

GetSourceOverride()

Returns the state of the Source Override flag in the current camera's structure.

GetCameraStatus()

This function returns the activity status of the "current" camera:

- CAMERA_STOPPED is returned if the destination has been reached.
- CAMERA_ACTIVE if not.

SetCameraMode(IDLE|PURSUIT_MODE)

This function controls the two operative modes of the Camera System:

- IDLE is used to halt all camera pursuit activities.
- PURSUIT_MODE is used as the normal and default camera operation.

GetCameraMode()

Returns the value of the current camera mode. See above for details.

SetWhichCamera(*thisCamera)

Passing this function a pointer to a CAMERA object will make that camera the current camera. All subsequent camera operations will affect this camera only. Leaving all other cameras completely alone and utterly helpless.

Offset Functions

These functions are useful to the extent that they can be used to temporarily manipulate the current camera by *adding to the value* of an existing variable.

SetLocaleOffset(angle)

Same as SetLocale() but applies an offset to that existing value. This value is ZERO by default.

GetLocaleOffset()

Returns the offset value for *Locale* being applied to the current camera (if any).

SetDistanceOffset(distance)

Same as SetTrackDistance() but applies an offset to that existing value. This value is ZERO by default.

GetDistanceOffset()

Returns the offset value for *TrackDistance* being applied to the current camera (if any).

SetAltitudeOffset(relative_altitude)

Same as SetTrackAltitude() but applies an offset to that existing value. This value is ZERO by default.

GetAltitudeOffset()

Returns the offset value for *TrackAltitude* being applied to the current camera (if any).

SetTrackTargetOffset(x, y, z)

Applies an offset to the track target position. This value is ZERO by default.

ResetOffsets()

Resets all of the above offsets to ZERO.

Locking Functions

These functions are (or will be) used extensively to temporarily lock aspects of the Camera System so they will not affect the view in their normal way.

For instance, you can apply a lock to the camera itself using the function SetCameraLock(ON) and until you unlock it, the camera will sit obediently in its current location. Meanwhile all other aspects of the Camera System will function oblivious to the camera's dilemma. Tracking of objects will continue, but now your camera acts as if its on a tri-pod. It can look – but it can't touch.

SetCameraLock(ON | OFF)

Prevents the Camera from moving from its location when set to ON. **Default is OFF.**

GetCameraLock()

Returns the state of the Camera Lock for the current camera.

SetTargetLock(ON | OFF)

Prevents the View Reference from being updated. **Default is OFF.**

GetTargetLock()

Returns the state of the Target Lock for the current camera.

SetLocaleLock(ON | OFF)

Prevents the Locale being updated. **Default is OFF.**

GetLocaleLock()

Returns the state of the Locale Lock for the current camera.

Miscellaneous Functions

These functions are not extensively used, but occasionally have been needed.

SetLocaleVelocity(velocity)

This function sets a constant locale velocity (should normally be zero). This value is ZERO by default.

AddLocaleVelocity(velocity)

Same as SetLocaleVelocity, but only a one-shot application used the next time UpdateCamera() is called.

GetLocaleVelocity()

Returns the current velocity applied to the current camera.

GetAverageVelocity()

Calculates and returns the velocity of the camera. The current implementation uses a distance approximation (max + 3/8 min), which historically has been quicker than square root of the sum of the squares for magnitude. This is probably a moot point on the PS2, but we shall see.

*GetTrackTarget(*x, *y, *z)*

Returns the coordinates of the track object.

*GetCamera(*x, *y, *z)*

Returns the coordinates of the camera.

StartShakeCamera()

Starts a camera shake, used for impacts and bumps to show the effect of the *camera* getting pounded.

Secondary Control Functions

Necessary tools for the alternate control method of the Camera System.

SetViewReference(x, y, z)

Access the decoupled point for target object, View Reference.

SetSurveillancePoint(x, y, z)

Access the decoupled point for the camera object, Surveillance Point.

Physics & Collision

Brief Overview

Physics System

The physics system is a 10+ degrees of freedom, rigid body system. Cars are modeled as single rigid bodies with wheel masses attached by springs, with additional springs providing roll support (this is additional support comes from a "sway bar" in actual cars). Cars will follow the laws of physics in their motion, interaction with the road, and collisions with other objects. Different road surfaces and different tires will have different coefficients of friction, which will effect the car's handling. We will also model differential power output from the engine to the wheels (a standard feature on 4WD cars), allowing us to further fine-tune the handling of the cars.

Risks - Medium

In order to interact with the environment correctly, the physics engine must be run at significantly greater than 60 fps. Assuming that a car's maximum speed is 120mph, we need to be able to run the physics engine at about 6 times per frame if we want to be able to resolve objects that are six inches wide. The physics engine is complicated, and this may involve a significant amount of processing. Further, if we need to run the full physics engine for each of the NPC cars (see Car AI section), the amount of processing required goes up by a factor of ten. We will *probably* end up running the physics on VU0, but whether the VU0 can handle the load, and whether this off loads enough processing from the CPU remains to be seen. If needed, we will drop down to a more basic physics model (incorporating many fudge factors for frictional forces) or run the physics engine at a slower rate.

There are also currently issues involved in calculating the frictional forces, but those should be resolved as the physics engine is built.

Collision Detection

Initial culling for collision detection will be based upon gems. An object in a gem will collide with object in that gem's visibility list. It will not collide with objects that are not in its gem's visibility list.

Individual objects will have collision hierarchies of either bounding spheres or bounding boxes. Different objects can use different collision geometry, as appropriate. Objects will have a flag to tell the collision detection module whether to base collision decision solely on bounding geometry, or to proceed down to the polygon level.

Collision detection for cars will be run at 6-8 times per frame, enabling us to resolve objects 6" wide when the car is moving at 120 mph. Collision detection for other, slower moving objects, will happen less often. This should be good enough to keep objects from going through one another, or significantly interpenetrating.

Risks –Low

There are two issues – (1) will the collision detection system run fast enough, and (2) will it be accurate enough. There always ends up being a trade off between the two. Accuracy comes from polygon-polygon intersections run extremely often, and speed comes from simple bounding box calculations, done occasionally. Since we can control both the rate of collision detection and the type of collision detection (polygon or box), we should have adequate control to make the trade offs correctly.

Vehicle model

At the most basic level, the car is modeled as a single rigid body with four smaller rigid bodies (a.k.a. "wheels") attached to it via damped springs (a.k.a. "suspension").

Although we may decide to have other moving bodies attached hierarchically to the car (for example, headlights that rotate up, or an antenna that sticks out the back), they will not affect the physics model.

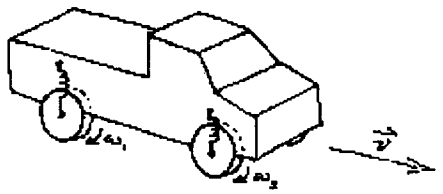


fig 1 - basic physics model

The car is given the standard six degrees of freedom – translation in x, y, and z, and rotation about the three axis. The wheels are allowed to rotate more or less independently, so each wheel has a separate rotational velocity ω. In addition, the front wheels pivot from side to side ("steering") and all the wheels move up and down in response to bumps in the road. All together this is 6+4+1+4 = 15 degrees of freedom, although the angular velocities of the individual wheels are somewhat coupled.

With regard to the competition – we have seen racing games that claim to use the angular momentum generated by the turning engine as a separate degree of freedom. To our knowledge, the only thing in a gasoline engine with much angular momentum is the fan, and we don't believe that is massive enough to contribute significantly to the angular momentum of the car. However, we are not an experts on engines, and anyone who knows better is welcome to enlighten us.

Engine vibration

It has been pointed out that the car should vibrate as the engine is revved. That's fine, but the vibration should not be part of the physics model per se. For the most part, the vibration is cosmetic – it doesn't really affect how the car steers, or how it flies through the air, or how it hits bumps in the road.

What effect the vibration does have is much smaller than the physical effects we are trying to model here. That doesn't mean that the car can't vibrate – it's easy enough to add a small offset to the position of the body at each frame. It just means that the vibration isn't part of the physics.

Suspension system

There are three main factors that influence the suspension:

- The spring constant
- The damping constant
- The maximum length of travel *(both up and down)* of the wheel.

If the wheel is at its maximum travel, additional forces in the max'd direction will be transmitted directly to the car body. For the first pass, we plan to make the spring constant, well... constant. However, in real life the springs are often made so that the spring constant increases as the spring is compressed, resulting in better handling. *Depending on time and performance issues, we may wish to make this adjustment.*

Unless someone can give a good reason for doing otherwise, the suspension system parameters – spring constant, damping constant, etc – will be the same for each wheel.

Body Sway

Related to the suspension system is a bit of hardware called the sway bar. The sway bar reduces body sway (which is produced by inertia when going around a turn) by adding a spring that counteracts the lateral differential compression of the suspension system.

In English, that means if the car heels over to the right, the sway bar pushes it back left. A good sway bar is essential to the handling of a vehicle with a high center of gravity.

Tires and Road Friction

There are three different kinds of friction associated with the road surface:

- Static friction
- Sliding friction
- Rotating friction

Static friction is the friction you need to overcome to get something to start sliding. Sliding friction is the friction associated with objects slipping over the ground, and rotating friction is the friction associated with rolling objects.

In general, F(static) > F(sliding) > F(rotating). We can model a variety of different tires and road surfaces by creating a simple lookup table with these values.

A final column in the lookup table would be used for all "non-tire" objects – like the car itself sliding along the road.

In real life, thin, wide tires deform less when going around a turn, giving the car better traction. In the game approximation, I'm not quite crazy enough to try to model this effect directly. We'll probably ignore this factor for the first revision of the physics engine, but depending upon time and performance issues, we may want to add it in later. I would suggest modeling it as a "tire performance coefficient", between 0 and 1, which would be weighted by the radial acceleration. You'd multiply this coefficient and the normal static friction of the tire/road combination together to get a new coefficient of static friction, which would in turn determine whether the wheel lost traction with the ground.

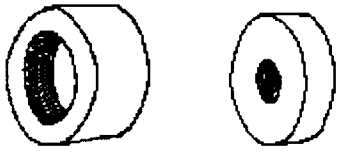


fig 2 - wide, thin tire vs narrow, thick tire

Flat Tires

Generally, if a tire goes flat and you insist upon driving on it, the rims cut the wheel to bits and you end up riding on steel. Without a tire, the wheel's effective radius is reduced, you lose a lot of traction with the road, and you probably start doing serious damage to the car itself.

Bumpers

Bumpers will not be part of the physical model. Collisions with the bumpers will be reported back to a damage assessment model (as will all collisions except collisions between terrain and tires). It is the responsibility of the damage assessment module to determine whether the bumpers can absorb the collision without damage, or whether there is harm done to the car.

Oversteer and understeer

Oversteer occurs when the car turns quickly and the back end swings out further than expected, causing the car to turn faster than expected (albeit with somewhat less control).

Understeer occurs when the front end swings out, causing the car to turn less quickly than expected.

Both these effects can be modeled by inertial "forces", and depend upon the weight distribution of the car. If the static friction of the road with the tires in the **back** of the car is not great enough to move the back end of the car without slipping, oversteer occurs. If the static friction of the road with the tires of the **front** of the car is not great enough, oversteer occurs.

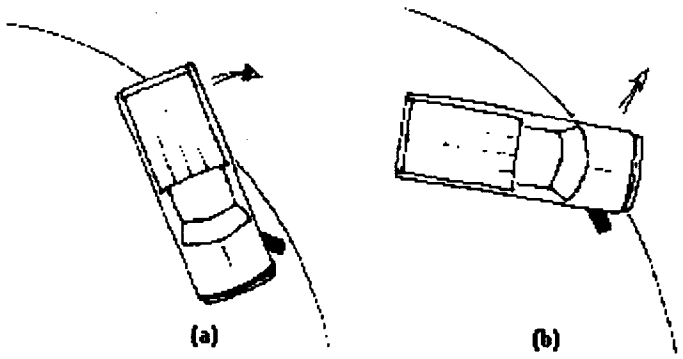


fig 3 - Oversteer (a) and understeer (b)

Engine Power and Differential Traction

The power that can be provided by an engine varies depending on the RPM of the engine. Every engine has its own Power versus RPM curve. For this game, we'll probably model that curve with a few points, and interpolate between them when necessary.

The power that the engine puts out is used by the wheels; causing the car to move faster, jump higher, whatever. However, the power available to each wheel is not necessarily constant, and is also not necessarily the same as the other wheels. This fact is extremely important in off-road racing, where different wheels may be subject to very different forces.

In real life, there are a couple of ways of applying different amounts of power to different wheels:

- Cutting power to wheels that are slipping while increasing power to the wheels that are gripping.
- Applying brakes to slipping wheels.

We can do either in our game. The effectiveness of this system (part of the car's "handling") will be reflected in how quickly the power loads can be rebalanced.

Collision detection

In discussing collision detection, we are going to differentiate between:

- Collisions between ordinary objects.
- Collisions between the tires and the terrain.

Collisions between ordinary objects are relatively rare, and will be calculated in the standard way – using bounding boxes or spheres first, finally proceeding to intersections between individual polygons.

Collisions between the tires and the terrain happen **all the time**. It's called driving. To help simplify things a little bit, we model the wheels as cylinders of some given radius and width, and intersect that cylinder with the polygons of the road.

- We first cull the possible candidates for collision by comparing their footprint with the footprint of the wheel.
- Then we check to see if the plane of the polygon is within the radius of the center of the wheel.
- Finally, we do the full intersection with any remaining polygons, figure out if any serious collisions have occurred *(penetration greater than some* ε*)* and adjust the suspension system accordingly.

Collision deltas, or Yes, We Really Do Have to Check That Often

One of the more difficult problems in collision detection is noticing when one object has **passed through** another object. At the first time slice the object is in front of the other object, and at the second timeslice it's behind. At neither time do the two objects collide, yet, something happened in the middle.

There are several methods for handling this problem:

- The first is brute force – we make the Δt's small enough that, compared to the size of the objects that are colliding, we don't miss the collisions very often.
- The second is predictive paths – we figure out where the object(s) would be if they didn't collide, and then extend bounding boxes around the entire path of the object. If the extended bounding boxes collide, we increase the time resolution to figure out exactly when and where the collision occurred. This method works best when colliding individual objects with individual bounding boxes.
- The third method is front/back differencing – first object was in front of the other object initially, but behind after the time step, a collision probably occurred. This method works best with collisions with static surfaces (like walls), or with collisions with objects that are much faster than the surrounding objects (like bullets).

The trouble with the second and third method for this type of game is that they're optimized for the non-colliding case. Most of the time the bounding boxes won't collide (or at least, there'll be only one collision within the time slice). Also, most of the time, the object won't have gone through the wall and we won't have to figure out exactly when and where.

Unfortunately, in a driving game over rough terrain, we're colliding with things on the ground all most all the time. Worse, the **lack** of a collision also has physical effects! The car doesn't just sail smoothly over a pothole. It drops down due to gravity, and possibly slams into the far rim of the hole. Neither the second nor the third method can account for this.

So we're back to brute force, at least for moving the car over the ground. Now the question becomes, "How often do we have to check?" The answer to that question really depends on the resolution you want when interacting with ground objects.

If you want the car to react fairly naturally to a pothole that is 2 feet wide, you want to check at intervals of significantly less than 2 feet. The exact timing depends upon the speed of the car.

Doing some quick calculations, maximum car speed = 120mph = 3 ft / (1/60sec), and if we want our resolution to be about 5 inches *(not unreasonable)* that's 8 times per frame. Clearly if the car is going slower we don't have to run the physics model and check for collisions as often, but that's about the upper limit. This values are again different with the addition of nitro.

Rollover and the problem of self-righting

In the interest of game play, cars should never get stuck upside down. (For some reason, players find "cockroach emulation mode" annoying.) Clearly, in the real world with real physics, cars do occasionally land on their backs. There are several possible schemes to get cars back on their wheels. I'm not stuck on any of them yet, and would welcome further suggestions. The fourth suggestion, "Hand of God" will probably be implemented as a fall back when all else fails.

- **Bug wiggle.** This is actually used in an existing game, Rally Cross. The top left and right controller buttons wiggle the prostrate car back and forth. Eventually it gathers enough momentum and flips over.
- **Invisible rubber band.** A "righting force" is applied to help push the car over. This is a bit more difficult than the option above, and may cause some motion artifacts at other times.
- **Predictive push.** When we detect that the car is flipping over, we make a guess as to where and how it's going to come to a stop. If the final position isn't to our liking, we add an additional little push or pull.
- **Hand of God.** The car is on its back. The angels sing. The car flips over.

Factors that aren't in the model

Because even next generation systems are never powerful enough, many effects that influence the motion of the car have been left out of the physics model. Here we list as many of them as we can think of, so that we really understand not just what we **are** modeling but what we **aren't**, and so that anybody who feels strongly that they should be included can get in a few cents worth.

These things have been left out:

- **Effect of wide-thin tires on traction.** As mentioned in the tire section, wide-thin tires hold their shape better on turns, resulting in better handling.
- **Tire pressure.** This effect is similar to the wide-thin tires effect. Under inflated tires deform under pressure, resulting in poor handling.
- **Torsion bars/Frame twisting.** Torsion bars are put between the struts of the vehicle, and resist the twisting of the frame. Since the model of the vehicle frame is entirely rigid, we don't need torsion bars.
- **Wind.** Although we'll add a frictional force proportional to the speed of the vehicle (i.e., wind resistance), we are not planning to model the wind itself. Nope, your top-heavy Toyota will not get blown over by a gust coming off the desert.
- **Effect of changes to the car's geometry.** As stated at the beginning, we are not going to change the mass, the center of gravity, or the inertia tensor because something inside the car moved, or because a fender fell off.
- **Springiness/sponginess of the ground.** The ground may provide more or less friction, but it is always pretty hard. What about mud?

Performance Analysis

The physics model for the cars in the game isn't overly simple. In fact, it's about as complex as the models used in the current generation of high-end PC racing games. Given that the game is supposed to be more of an arcade game than a simulation, one might ask if we need a physics model that is this complex. To that question, we would answer YES, for a number of reasons:

- **Because it will be expected.** As the processing power available goes up, the expectations of the players go up. Already, one of the main complaints people have about racing and driving games

is that the physics doesn't feel good enough. People didn't complain about physics back in the Sega Genesis days. Going forward, the complaints are just going to get worse.

- **Because it's an off-road racing game!** This game is all about bumping over things, getting air and bottoming out the suspension. You can't do that in a convincing manner without paying a lot of attention to the physics of the model and the interactions between the model and the road. Maybe you can get away with a simpler model in a game like *Driver*, but you can't here.

Given that we want a good physics model, the next question is: "Can we afford it?" Now, consider. Physics models with this degree of complexity are already being run on today's PCs. **They** do it. Yes, the physics sections will show up on the performance analysis charts. But no, it shouldn't prevent the game from being fun. We are considering off-loading a lot of the physics calculations to the VU0, running in independent mode. It seems like an ideal thing for that processor to do.

However, here are couple of suggestions on how to reduce the processing power needed by the physics engine, if we find it necessary:

- **Not all cars are created equal.** Cars that cannot be seen by the player don't have to follow the same laws of physics. This is the "If a car hits a tree in a forest and no one hears it, do I really care?" dilemma. We can use much simpler statistical methods to determine the paths taken by offscreen cars, and the damage inflicted upon them and by them.
- **Reduction of complexity.** If necessary, we can model such effects as oversteer and understeer, wheel spin and skid in look up tables instead of calculating forces. If necessary, we can eliminate differential wheel power, and the effect of the angular momentum of the car's wheels on the torques experienced by the car itself. We can also reduce complexity dynamically as required.

In the text of this document, I've also included a couple of suggestions on ways to make the physics model slightly **more** complex, should we decide we can spare the extra cycles.

Some math

The actual physics in this model is derived from standard Newtonian mechanics: $F=ma = dp/dt$, $T=I\alpha=dL/dt$, momentum conserved otherwise.

Forces come from several different places:

- The engine, acting upon the wheels.
- The brakes, acting upon the wheels.
- Collisions with objects or the road.
- The springs, acting upon the wheels and the body.
- Wind resistance, acting upon the body and proportional to its velocity.
- Our friend, gravity.

We intend to use Euler's method to approximate the results of the differential equations. This method is quick and easy, although not very accurate. However, this is a game not an engineering simulation, and we believe that the inaccuracies caused by the solution method will not be visible to the player.

Euler's method also suffers somewhat from instability – if the springs used in the simulation are "stiff" with respect to the time steps used to evaluate the differential equations, huge internal forces can be generated that cause the model to behave in all sorts of undesirable ways. To avoid this problem, we shall be keeping the springs in the model to a minimum, and are planning to use impulsive forces rather than stiff springs to model collisions.

Although stiff springs probably produce a better model, we'd like to make sure that a) We can run the simulation **and** play the game at the same time, and b) Cars don't suddenly start leaping tall buildings at a single bound.

Pertinent Structures

The physical model of a car consists of a rigid body (the car body), which is tied via springs to four wheels. The springs on the wheels are constrained to move only up and down, as seen in car body coordinates.

Structure holding initial dimensions of the vehicle

```
typedef struct physCarDim {
    int         bodyW, bodyL, bodyH;                    Body width, length, and height.
    int         cabW, cabL, cabH;                       Cab offset, width, length, and height.
    int         engW, engL, engH;                       Engine offset, width, length, and height.
    int         dtrW, dtrL, dtrH;                       Drive-train offset, width, length, and height.
    int         bodyOSx, bodyOSy, bodyOSz;              Offset of body from origin.
    int         cabOSx, cabOSy, cabOSz;                 Offset of cab from origin.
    int         engOSx, engOSy, engOSz;                 Offset of engine from origin.
    int         dtrOSx, dtrOSy, dtrOSz;                 Offset of divetrain from origin.
    int         bodyM, cabM, engM, dtrM;                Mass of above.
    int         axleR, axleF;                           Positions of the axle.
    int         wheelR, wheelW, wheelM;                 Wheel info.
    float       gear[6];                                Gear ratios.
    float       differential;                           Differential ratio.
    int         powerCurve[9];                          Power in hp, at x1000 rpm
    int         torqueCurve[9];                         Torque in ft-lbs, at x1000 rpm
} physCarDim;
```

Structure holding physics properties of the vehicle

```
typedef struct physCar {
    ps4Matrix   lwMatrix;                               Position and rotation of car.  Measured from center of mass.
    ps4Matrix   wlMatrix;                               Inverse to the above.  From world to local coordinates.
    ps3Matrix   bodyPMI;                                From body coordinates to principal axis coordinates.
    ps2Tensor   inertia;                                Inertia tensor.
    ps3Vector   velocity;                               Velocity of the center of mass, in world coordinates.
    float       posX, posY, posZ;                       Car position in the world, measured from center of mass.
    float       roll, pitch, yaw;                       Car angular coordinates, in human understandable terms! see also lwMatrix
    ps3Vector   angVelocity;                            Car angular velocity around center of mass.
    float       cmx, cmy, cmz;                          Center of mass, in car coordinates.
    float       width;                                  Width of car collision geometry – when do we get the rest of the car?
    int         mass;                                   Mass of car
    int         gear;                                   Gear car is currently in.
    int         tireM;                                  Mass of tire of car.
    int         tireX;                                  x value on side of car where tires grow XXX need something else here!
    float       axlePosZ[2];                            Neutral position of the tires, Z axis
    ps3Vector   tirePos[4];                             Position of the four tires
    float       tireOmega[4];                           Angular velocity (around the axle) of the four tires.
    int         axleF, axleR;                           Position of the axles.
    physCarDim  dim;                                    Car's physics dimensions.
    float       tireR, tireW;                           Wheel/tire dimensions.
    msgTire     tireType;                               Type of tire – referencing enumeration.
    float       tireAngle;                              Steering.  Angle that the front tires are moved through.
    float       axleZ;                                  Neutral position of the axle of the wheels.
    float       tireFRvel, tireRRvel, tireFLvel, tireRLvel;  Velocity of the tires to or from the body of the car.
    float       shockK, shockD;                         Shock absorbers, spring constant and damping factor.
    float       shockMaxX;                              Maximum amount of travel.
    float       brake, gas;                             Amount of gas and brake applied, 0 -> 1 (is this really part of the car?)
    float       maxBrake, maxGas;                       Maximum amount of force that the brakes and the engine can generate.
    float       gearRatio[6];                           Gear ratios.
    float       differential;                           Differential ratio.
    float       torqueCurve[9];                         Torque in ft-lbs, at x1000 rpm
} physCar;
```

What's left out temporarily

Sway bar, differential engine power, flat tires, rpm, flags to tell whether the car is skidding, has wheels off the ground.

What's left out permanently

Toe in/out, angular momentum of wheels, effective cross-section of drag *(not the same as wind resistance).*

Networking

Networking will be implemented by building a socket-style interface on top of whatever networking protocols Sony gives us. Both connected (TCP) and connectionless (UDP) types of sockets will be supported.

Risks - High

Networking on the PS2 is a great unknown. We don't know what network they're planning to run over, what the level of support for the networking protocols is, etc. Unfortunately, it is very hard to work networking into a completed game; we really want to be working on it from the start. Without the necessary information from Sony, networking is likely to be implemented in a non-optimal way, or squeezed out of the schedule.

Brief Overview

The Network Features describes a basic architecture of network engine that may be used for building a vatriety of games with network capabilities or local/global networks oriented. This document attempts to detail basic principles of engine, general interfaces and optimization solutions.

Goals

This network engine scheme has 2 primary goals:

- Get the current project out on time as full featured as possible. While this goal is obvious, it is stated here for completeness.
- Be used for future Press Start/Interplay game development.

Block Diagram

The following block diagram gives a high level view of the network engine:

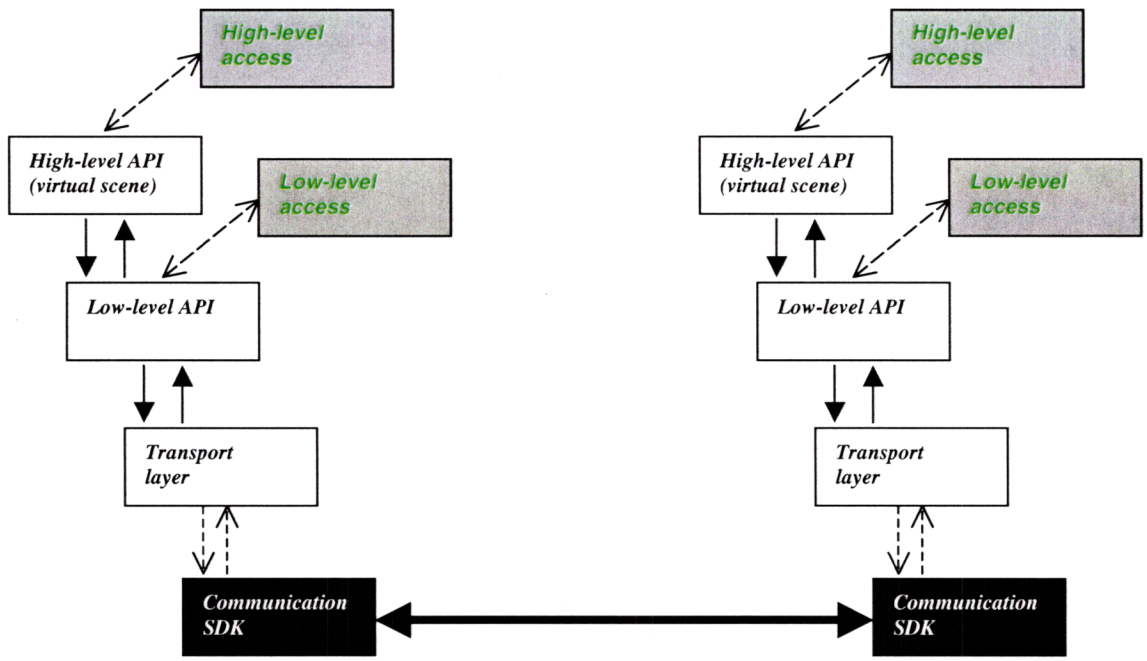


Figure 1 Network Engine Diagram

Description of Figure 1

Solid bold arrow

Communication environment *(direct cable, local area network, wide area network, etc).*

Solid thin arrow

Links between internal components of network engine.

Dotted arrow

Links between network engine and external environment.

White blocks

Internal components of network engine.

Grey blocks

External environment.

Black blocks

Third party software.

Third party software – communication SDK

Third party software is used as the low-level communication layer of the network engine. Unfortunately we still don't have any descriptions of communication SDK for PSX2, but we hopes this SDK will provide hardware-independent access to network. Hardware classification:

- Direct cable
- Modem
- LAN adapter

Transport layer

This layer needs for supporting of network protocols. Transport layer could be an original or third party software (such WinSock SDK or Sony Libraries).

Protocols classification:

- TCP/IP
- Sony proprietary protocols

Low-Level API

Low-level API provides easy access to capabilities of network engine. The level could be used for direct connection between users or as client part for third party server.

```
ERRNO    OpenChannel ( CHANNEL& chnl );
ERRNO    SetMode ( MODE mode );
STATUS   GetStatus ( CHANNEL chnl );
bool     Connect( ADDRESS address );
ERRNO    Listen ( int num_of_port );
ERRNO    Accept ( CHANNEL chnl );
ERRNO    Send ( CHANNEL chnl, DATA data, UINT size );
SIZE     Receive ( CHANNEL chnl, DATA& data );
void     CloseChannel ( CHANNEL chnl  );
```

Channel

```
#define     CHANNEL     void*       // channel for connection
```

Channel is an abstract connection for the network engine. You can send and receive data using a channel, and is a full-duplex connection.

Error Code

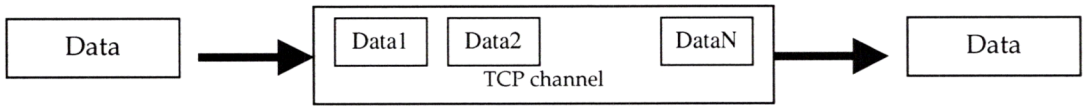
```
#define     ERRNO       int         // code of error
```

Type of connection

```
typedef enum
{
   MODE_TCP,                        // safe connection
   MODE_UDP,                        // fast connection, data lost possible
   MODE_UDP_CORRECTION              // fast connection w/error correction
} MODE;                             // type of connection
```

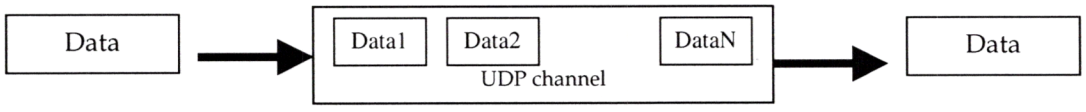
Safe connection

Error correction provided by network protocols (for example, for TCP/IP you need to use TCP transport protocol). The protocol guarantee that package will be send and receive correctly and in the same order. The connection is safe, but slow. Recommended for networks with unsafe interconnection (modem lines, networks with low bandwidth etc.).



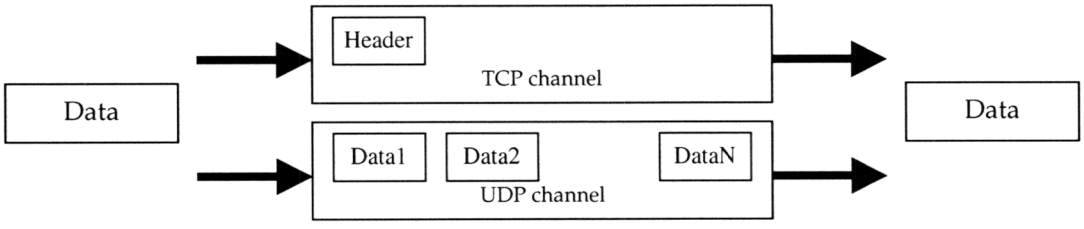
Fast connection

Connection without error correction (for example, for TCP/IP you could be use UDP transport protocol. Some packages could be lost (depend from type of network). The connection is fast, but unsafe. Recommended for safe networks with high bandwidth.



Fast connection with error correction

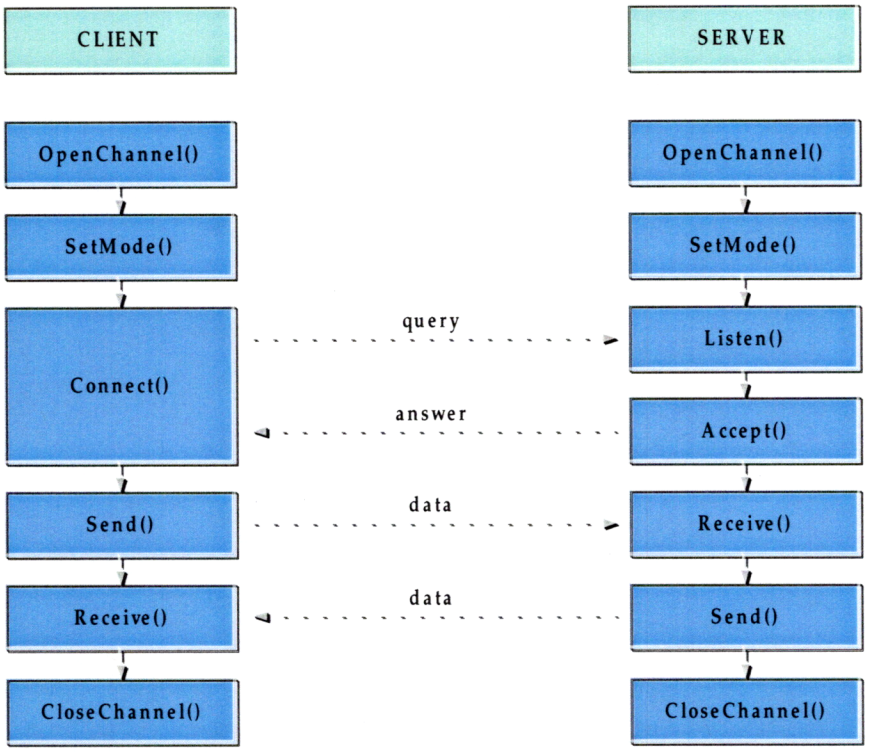
original type of connection. Based on combination of transport protocols (TCP and UDP for TCP/IP model). Recommended for unsafe networks.



Status of connection

```
typedef enum
{
   STATUS_READ,
   STATUS_WRITE,
   STATUS_ERROR
} STATUS;                          // current status of channel
```

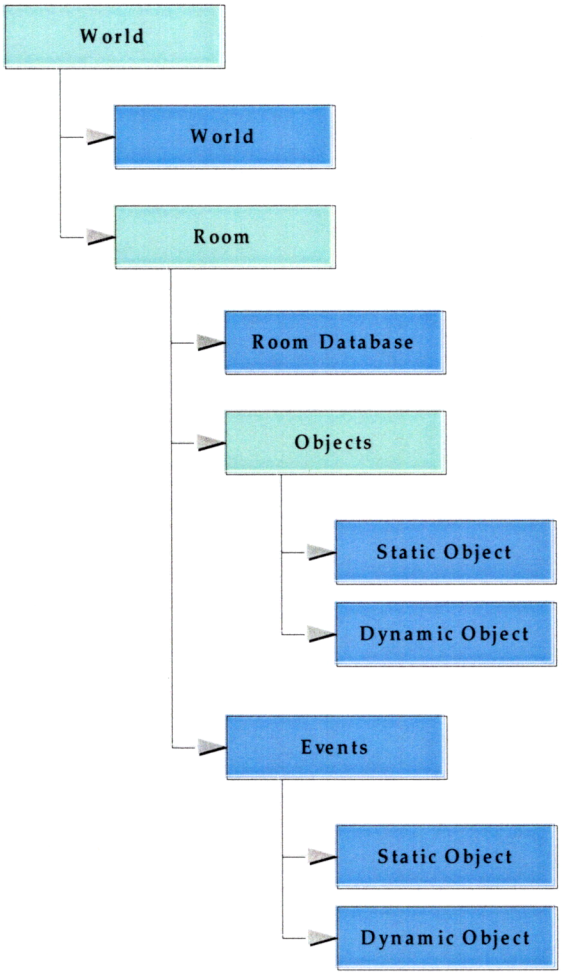
Low-level API usage



High-Level API

High-level API provides an access to client-server capabilities of network engine. The level could be used for organization of multiplayer game in the local/global networks.

Object model of high-level API



High-level API includes three levels:

- World level
- Room level
- Object level

World level includes rooms' collection and global world database. If you have an access to the level, you have an access to the members only. Rooms' collection keeps an information about all rooms. World database store global world data i.e. global counters, global time manager, administration information, etcetera…

Room level includes objects' collection, events' collection, and room database. If you have an access to the level, you have an access to the members only. Object's collection keeps an information about all objects. Event's collection keeps an information about all events. Room database store global room data i.e. local counters, local time manager, users information…

Object level provides an access to all objects' properties and methods. Also you can work with user's events – send and receive it.

Types of object:

- **Static** – objects which exist always during lifetime of room. The static objects can be modified, but no add or remove.
- **Dynamic** – the objects can be added and removed without restriction. Any modifications available too.

Type of events:

- **System** – the common events for all users.
- **User's** – each user can register event and use it, the events will be available for the same users only *(reserved for future versions)*.

All objects have unique identifier. For TCP/IP networks, the identifier has following structure:

IP address # local number

Local numbers generates inside network engine automatically.

Client-server architecture

High-level API bases on original client-server architecture. The architecture support flexible connection model without restrictions for number of users and order of connections.

Features:

- Unfixed server component;
- Unfixed number of users;
- Data flow optimization;
- Data protection.

Server component

Features:

- Can be used as part of client application or independent application
- Support of arbiter functionality
- Provide users management
- Storage for global data (counters, results, etc)
- Recovery of broken connections
- Checking of incoming packages
- Spam blocking

Server's arbiter functionality (sample for two clients)

	t_1	t_2	t_3	t_4	t_5
User1	P_{11} >> Server				P_{11} << Server P_{21} << Server
User2		P_{21} >> Server			P_{11} << Server P_{21} << Server
Server		P_{11} << User1	P_{21} << User2	P_{11} >> User1 P_{21} >> User1 P_{11} >> User2 P_{21} >> User2	

t_n time;
P_{nm} data package, n – number of user, m – number of package;
>> send operation;
<< receive operation.

Client component

Features:

- User's data storage;
- Checking of incoming packages;
- Spam blocking.

World level

```
typedef struct
{
      CHAR* name;         // world name
      ROOM* rooms;        // rooms collection
      VOID* data;         // world's internal database
} WORLD;

typedef struct
{
      BOOL  isRef;        // reference status
      CHAR* path;         // path to object
} REF;

typedef REF WORLDREF;
```

Server side interface

```
WORLDREF    CreateWorld( WORLD* world );
```

Parameters: *world* – pointer to WORLD structure.
Return value: reference to the created world.

```
VOID        CloseWorld( WORLDREF refWorld );
```

Parameters: *refWorld* – reference to the existing world.
Return value: none.
Comments: the creator of world can close it only.

Client side interface

```
BOOL        EnterToWorld( WORLDREF refWorld, CHAR* login, CHAR* password );
```

Parameters: *refWorld* – reference to the existing world, *login* – user's name, *password* – user's password *(login and password needs for access protection, could be not used in safety networks)*
Return value: true, if the world still exist.
Comments: all your added rooms will be available for another users after successful enter.

```
VOID        ExitFromWorld( WORLDREF refWorld );
```

Parameters: *refWorld* – reference to the existing world.
Return value: none.
Comments: all your added rooms will be deleted from world after exit.

Common

```
WORLD*      FindWorld( CHAR* name );
```

Parameters: *name* – name of world.
Return value: reference to the world; if not found, return NULL.

Room level

```
typedef struct
{
      CHAR*   name;          // room name
      OBJECT* objects;       // objects collection
      EVENT*  events;        // events collection
      VOID*   data;          // room's internal database
} ROOM;

typedef REF ROOMREF;
```

Server side:

N/A

Client side:

```
ROOMREF CreateRoom( ROOM* room );
```

Parameters: *room* – pointer to ROOM structure;
Return value: reference to the created room.

```
ROOMREF FindRoom( CHAR* name );
```

Parameters: *name* – name of room;
Return value: reference to the room in the current world, if not found, return NULL.

```
VOID CloseRoom( ROOMREF refRoom );
```

Parameters: *refRoom* – reference to the existing room;
Return value: none;
Comments: the creator of room can close it only.

```
BOOL EnterToRoom( ROOMREF refRoom );
```

Parameters: *refRoom* – reference to the existing room;
Return value: true, if the room still exist;
Comments: all your added objects will be available for another users after successful enter.

```
VOID ExitFromRoom( ROOMREF refRoom );
```

Parameters: *refRoom* – reference to the existing room;
Return value: none;
Comments: all your added objects will be deleted from room after exit.

Object level

```
typedef enum
{
      OBJECT_STATIC,     // static object (see description in obj model of high-level API)
      OBJECT_DYNAMIC,    // dynamic object (see description in obj model of high-level API)
} OBJECT_TYPE;

typedef struct
{
      CHAR*       name; // room name
      OBJECT_TYPE type; // object's type
      VOID*       data; // object's internal data
} OBJECT;
```

```
typedef REF OBJECTREF;
```

Server side:

N/A

Client side:

```
BOOL  AddObject( OBJECT* object );
```

Parameters: *bject* – pointer to object structure.
Return value: true, if object with the same name doesn't exist in the current room; false, if exist already.
Comments: the function send request about creation of object at the room. The event OBJECT_ADDED will be available via **GetEvent()**.

```
BOOL  DeleteObject( OBJECTREF refObject );
```

Parameters: *refObject* – reference to existing object.
Return value: true, if current client is owner of the object and the object still exist in the current room; false, if object has another owner or does not exist.
Comments: the function send request about deleting of object from the room. The event OBJECT_DELETED will be available via **GetEvent()**.

```
OBJECTREF  FindObject( CHAR* name );
```

Parameters: *name* – name of object.
Return value: reference to object in the current room; if not found, return NULL.

```
BOOL  ChangeObject( OBJECTREF refObject );
```

Parameters: *refObject* – reference to existing object.
Return value: true, if current client is owner of the object and the object still exist in the current room; false, if object has another owner or does not exist.
Comments: the function send request about changing of object at the room. The event OBJECT_CHANGED will be available via **GetEvent()**.

```
BOOL  GetEvent( EVENT &event, TIME &time );
```

Parameters: *event* – event structure; *time* – local room time.
Return value: true, if at least one event in the events' line, false if not. If true, new event will be available via *event* parameter and local time will be available via *time* parameter.

Events

```
typedef enum
{
        EMPTY = 0,              // no params
        OPEN,                   // no params
        CLOSE,                  // no params
        OBJECT_ADDED,           // param1 - OBJECTREF refObject
        OBJECT_DELETED,         // param1 - OBJECTREF refObject
        OBJECT_CHANGED,         // param1 - OBJECTREF refObject
} EVENT_TYPE;


typedef struct
{
        EVENT_TYPE type;        // type of event (see EVENT_TYPE description
        CHAR*      name;        // name of event - reserved for future versions
        VOID*      params;      // pointer to event parameters, may be NULL
} EVENT;
```

Menu/UI Implementation

A game menu is just another scene (or level) in the world. By taking this approach, our menus may be a very simple collection of text objects that appear on a background, or they may be as complicated as having a garage setting, where you pick items like tires, engines and the like by selecting them from items in the garage. As an extreme, you may even wish to have a chorus line of mice dancing to the garage radio in the menu scene, and have a particle effect of smoke coming up from a cigarette in the garage. Menu complexity is only limited by the limitations of the rest of the engine specification.

Menus are constructed using Menu AI objects that are built in 3D Studio or manually constructed in a text file for testing. Artists build a "3D menu scene," which may contain any object that the normal engine is capable of processing. By selecting that the AI type of an object is "menu", the artist is presented with options for the objects that allow them to set the "focus" to the input controller, as well as define what it's action is if the user selects it – such as "start game", "pause", or "pick truck type."

Menus are not limited to only menu objects. A menu is a separate scene which either overlays *(on top of)* or replaces the current scene.

The differences between a menu and a normal scene are:

- Menu AI objects are used where selectable options are wanted.
- Overlay menus (on top of a scene) must coexist in the world and be positioned in front of the camera in order to be seen.
- Control of the player's controller is taken over by the menu.
- Some or all of the normal game logic must "freeze" during a "pop up" menu. Sony standards will be implemented, as is normal.

Adding Menu AI to an object in a scene

As with any object in a scene, the artist places a "controller" node on the object. One of the characteristics of the controller node is which AI script is associated with the object. When the artist indicates that an object in a scene is a "menu scripted object", the options for a menu object are made available to the artist.

Examples of the options of a menu object are:

- Select Level
- Start Level
- Restart Race
- Select Truck
- Initial focus (which object is selectable when the menu first pops up).
- Next menu item to get focus
- Previous menu item to get focus

Controlling a menu during run time

Since menus are just another scene, all the normal game logic is used to control a menu. All objects are the same type of 3D objects that are used in the rest of the game. The following are exceptions to the normal game logic. These exceptions are not an issue where the menu is actually a stand-alone level, such as the screens that select which track you will race on before the race starts.

Overlaying a menu

If the menu is an overlay type, such as a boring list of text that appears like a pop-up window, then extra logic is required by the running game. The game must be paused, control reprioritized to run the appropriate menu tree (see controller priorities), and after the selection is made the game must be un-paused (or whatever the menu option dictates). In an overlay menu, the objects must also position themselves in front of the camera.

Taking control of the player's game controller

During the time that a menu is present, the normal scene AI is not ran due to a priority change. Since the controller input is scanned by the AI, control is automatically removed from the player to control the normal game. At the same time, the menu objects which have their own AI are now evaluating the game controller for input.

Passing menu selections back to the engine

When a menu is displayed, a message is sent down the controller tree telling each object that a menu is being shown. This allows the controller labeled with "starting focus" to be the only menu AI object excepting game controller input. The message that starts the menu may also override the focus item id if so desired, such as when backing out of a chain of menus where the last object selected should still be selected when the "menu stack is popped".

The user may then make selections such as "next menu item", which will change the focus to the next menu item (which the menu object stores). If the user presses the "accept" key to select a menu item, then it depends on the attributes of the menu item as to what the action is. If the menu item had the "start game" attribute, then the selected menu object would call the start game function (part of programming the menu object AI). If the menu item had the "select level" option, then the AI would do that call, which may be to update the menu item's text, or to show a new 3D object such as a track.

Menu object text

Text on a menu object may take two forms. The first is that it is geometrically drawn, where it is not text at all but a 3D object. The second form is a 3D object with a "text texture" on it. This is a texture where the engine places text in the texture. The texture is already mapped to the object, so the text appears. For the items that have "text textures" on them, the text will come from a file that contains all localization text within it. The Menu AI object will have the intelligence to be able to either scale the texture or the menu object depending upon a flag the artist may set when placing the menu object in the world.

Risks – Low

The only foreseeable risk is a frame rate problem if a menu becomes extremely complicated geometrically, and is meant to overlay (while displaying) any area in the current level. Since the levels will be filled with geometry until total load breaks the 60hz limit, overlay menus will have to remain simple. If this problem occurs it will only happen in an overlay menu, and it can be fixed by simply reducing the complexity of the menu (to the worst case of just text selection).

Input Handling

The default input device is the PlayStation analog dual shock controller as it will be shipped with the system. We will be polling the controller(s) multiple times per frame so that we can respond quickly to user actions. *When you're traveling 120mph, it counts.*

We plan to create an interface between the stream of low level data coming in and the high level game commands – "Accelerate", "Turn right" etcetera. The game engine will be able to simply look at the high level commands, rather than trying to interpret the low level data itself. This will allow us to easily change the mappings between sequences of key presses and high level commands, or even to move to a different controller.

Force feedback will be supported using the functions provided by Sony just as they were supported with the PSX. All controllers supported by the PSX for racing games (including analog devices, dual shock, etcetera) will be supported. All Sony Standards at the time of publication nearest submission will be observed.

Risks – Low

It's the same device we've used in past PlayStation games, and is unlikely to present many unknowns or problems.

Art Tool Chain

Brief overview

The 3D tool chain describes a common set of tools and processes that may be used for building a variety of games based upon a 3D scene-based world. This document attempts to detail not only the required tools to provide a smooth and expedient art/design production cycle, but the process as well, so is set up in an annotated form.

The art tool chain comes in four pieces:

- Press Start's 3D Studio Calliope™ plug-in, our object and world builder.
- Texture converter, from TARGA to whatever we decide to store textures in
- Animated texture converter, from TBD animation package (such as Debabelizer Pro) to proprietary format.
- Binary converter, from output of 3D Studio w/plug ins to proprietary format.

Of these four, the plug-ins to 3D Studio and the Binary converter are the most involved, and will be described here.

Calliope™ plug-in

The idea behind Calliope™ is to give the artist or production assistant maximum control of the action of the objects in the game. For instance, instead of the artist coming to the programmer and telling him/her that "this car ought to have this type of tires", Calliope™ will read a car-characteristic template when an object of type "car" is created. Calliope™ will display a dialog box on the screen containing, among other things, a pull down with options for the type of tire. The tire-type information will be saved as part of the instance of that car object, and can be loaded into the game without intervention of the programmer.

The feature described above is the "object template". The full list of features is described below:

- **Object template:** Allows artist/PA to input object-specific information that is later used in the game.
- **Minimum # of polygons:** Since we will be using a progressive mesh for many of our object LODs, we allow the artist to specify the minimum number of polygons to which the object may be decimated.
- **Gems:** These space-partitioning regions must be created by hand. Although hand building is tedious, it only has to be done once, and should give us a more efficient structure than standard BSP schemes. The plug in allows the artist/PA to both define the gems and to define the list of other gems visible from the gem being modified.
- **Sound events:** Although 3D Studio does not support sound, we can allow it to support sound events in animations. This plug in looks for "sound notes" during an animation sequence. These sound notes contain the name of the associated sound file, the volume at which to play the sound, the pitch, and other sound characteristics. They also define where a sound begins and ends during the animation.
- **Export inhibiter nodes:** Some objects in the 3D Studio database are used for reference only, and should not be exported (for instance, a gun in a character's hand, where the artist is trying to get the arm position of the character correct for holding a gun, but the type of gun he/she is holding changes depending upon in-game parameters.) Inhibiter nodes prevent any geometric objects below them in the hierarchy from being exported.

- **Collision spheres and/or collision rectangles:** The Artist or PA will be able to attach collision geometry to a node. The collision geometry will be hierarchical, with leaves containing lists of associated polygons. Collision geometry for a parent object is assumed to contain collision geometry for sub-objects.

The Calliope™ plug-in to 3D Studio will be set up to output the following four files:

- SDF – Scene Description File: Contains gem hierarchy info.
- GDF – Gem Description File: Contains the actual gem data structures, visibility lists, and references to instance data. Also contains object templates, and object property information, but not the geometry or animation data.
- MDF – Model Data File: Contains actual geometry data for one object template. Model data includes triangle based models, lights, collision geometry, trigger objects, and splines for path finding.
- ADF – Animation Data File: Contains animation information for the objects in the associated MDF. The MDF and the ADF contain the same object hierarchy.

Binary Converter

The output of 3D Studio (with Calliope™ plug-in) is ASCII data. We do this for a number of reasons:

- It's human readable, and we should be able to quickly find any problems in the output
- It's game and platform independent, and can be reused for any other game.
- This also allows us to hand-generate data to proceed with engine development until the tool chain is complete. *An additional step, the Binary Converter, is required to put the data into final, PS2 specific format.*

Clearly, the Binary Converter takes the output of 3D Studio (the geometry, region, animation and template data) and converts it into our own format. However, it also does some non trivial processing, explained below.

In addition to the standard conversion, the binary tool is responsible for the following:

- Creation of progressive meshes for LODs
- Creation of shadow maps.
- Division of region leafs into octrees.
- Error checking of data with meaningful error messages that allow the artists to find problems in the data without consulting an engineer.

Risks – Medium

This is a big task, and many things depend upon it. The risk here is not so much whether we will be able to create this tool, but whether unexpected difficulties creating the tools will significantly affect other parts of the schedule. To limit this risk, certain critical tasks (physics, collision detection, some rendering and car AI) will be started without the tool chain in place. This will cause some duplication of effort and some extra integration work, but will ensure that the project keeps moving. The binary tool is already in the state of reading the data required to build scenes with characters (without animation data or spline information as of yet). Although a tedious process, we will be able to read in hand-generated scenes and models without textures as soon as the engine is capable of reading the output of the binary converter.

Goals of the tool set

This tool set scheme has 3 primary goals.

- Get the current project out on time as full featured as possible. While this goal is obvious, it is stated here for completeness.
- Be used for future Press Start 3D game development. Every attempt has been made to insure that the tool chain only suffers from one dependency, which is that this tool chain makes worlds and characters for 3D worlds. While this design will work great for games such as our target, as well as sports arena games, first and third person shooters and the like, it would not be suitable for games such as Command and Conquer, or Starcraft.
- To provide the artist/designer with a new level of creative freedom, while offloading workload that has traditionally been assigned to programmers, such as adding sounds to an animation or manipulating AI characteristics.

Block Diagram

The following block diagram gives a high level view of the tool chain up to the point where the scene and character data (including geometry, animation, and texture data, as well as game specific data) are exported into a binary format suitable for the current PlayStation 2 target.

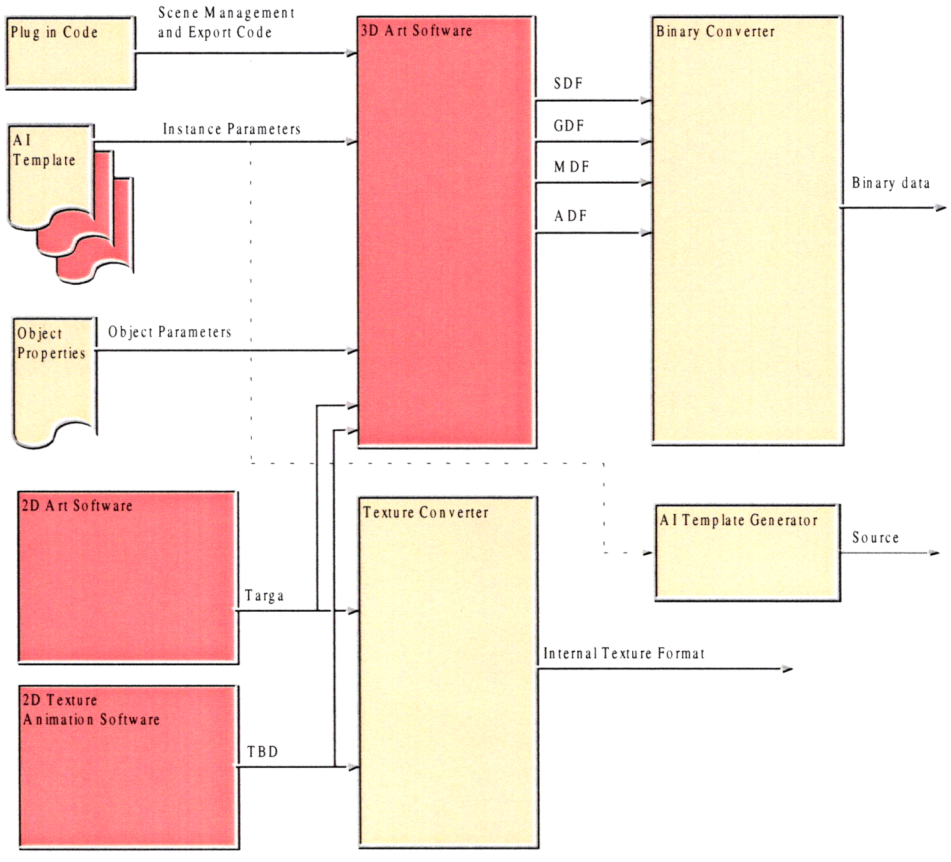


Figure 2 Tool Block Diagram

Overall description

The block depicts software that Press Start will develop in bold outlines, while third party art software is shown with a thinner boarder. The blocks titled "AI Templates" and "Object Properties" are text description files, and are explained below.

Third party software

Third party software is used as the core of the content development tools.

3D Art Software

Press Start is using 3D Studio MAX 3.0 to build level models (land and such) as well as character models and animation. Maya is also used to build some of the scenery, but is exported to 3D Studio MAX 3.0 for final integration and exporting to binary form for the engine.

2D Art Software

Press Start is using Photoshop and Debabelizer Pro (PC) as primary 2D art tools for making textures for the 3D environments and characters. Image data may be stored in a Photoshop native format to preserve features such as layers. Texture data exported for the Binary Converter will be in Targa format. Targa is used due to the availability of format documentation required to manipulate the textures into our internal format, as well as it maintaining a true 32 bit RGBA image.

2D Texture Animation Software

2D Animation software is required to support art effects like movie screens on the side of blimps, water effects, movies in windows during menu selections and the like. Art tools such as Animator Pro could fill most of these requirements. Selection of Tools for this task and the export format are TBD.

Audio Software

Audio Production software is required to create, edit, and process audio source files into a format useable by the tools and game engine.

Project developed software

This section lists the software requirements to transform the third party software into a tool suite for development of 3D worlds and characters.

Developed Audio Software

This section lists the software requirements to transform the third party audio software into a tool suite for development of audio. These tools must manage all of the different data types used to produce audio. The goal is a system that does not require re-building (compile and link) code whenever a sound is added, removed, or changed.

A tool will create a text file, "Master Audio List", with all information necessary to play the audio. The names of each sound will supplied for use by the 3D Art Tool as a list of available sounds for sound events. Please see section 5.5.4.

The "Master Audio List" will be loaded by the game at runtime whenever the audio is initialized. Information from the list will be used to play all sounds in the game engine.

A tool may be needed to convert the "Master Audio List" into a binary or compressed format for use by the game engine.

Whenever changes are made to the sounds a new "Master Audio List" will be made available to the 3D Art Tool and the game engine.

Calliope™ – art tool plug-in

Press Start is currently building the 3D Studio MAX plug-in (named Calliope) for use in building both scene data and character data along with animation. The following paragraphs give a description of the individual subsections of the plug-in.

Gems

Gems are used as a form of space partitioning. Unlike such schemes as the BSP (Binary Space Partition) tree, which is generated algorithmically, Gems are placed in the scene levels by an artist. While the obvious down-side of this is that manual labor by the artist is involved, there are benefits that outweigh the extra work.

Refer to Gems sections later in this section and in the Scene manager for more information.

Hierarchy Inhibitors

Hierarchy Inhibitors are a MAX helper object that is used to place within a hierarchy of objects to inform the exporter not to export the inhibitor or any object that is a child to the inhibitor. One use of this is to aid the animation artist. The animator may have objects that the character being worked on holds, such as a rifle for a shooting animation. When the animation is exported, the rifle is not intended to be a part of the character, so by placing an inhibitor on the hands of the character and placing the rifle on the inhibitor, the rifle is not exported.

Collision Spheres

Collision spheres are objects that may be placed on a model for use in collision checking in the engine. These spheres are normal objects in MAX in that they may animate with the model, but they display no geometry (except in MAX, where they display a spherical object that is distinctly different from the sphere and geosphere MAX objects so they may be readily recognized by the artist.

LOD Minimum Vertex List

The minimum vertex list object is a MAX vertex selection object that is used to select the vertices which can not be removed by the engine during LOD reduction. These are manually set by the artist. The tool works and acts just like the normal vertex selection tool in MAX.

Animation

The 3D art tool allows many types of animations to be exported with a model's animation data. The following section enumerates what animation types are used for export, as well as the format that they are export in. All of the exported animation data must generate key-frame data within MAX that can be viewed within MAX's "track view". Animation that relies on algorithmic plug-ins (such as Character Studio) may not be used unless that data is processed into data that will still work when the plug-in is removed. The reason for this is that Press Start does not intend to build a version of Character Studio or other plug-ins into the engine.

Position, Rotation, Scale

Position and scale are performed with the default bezier position and scale controllers in MAX. Rotation is performed with the default MAX TCB rotation controller. Use of any other type of position, rotation or scale controller will generate an error during export identifying the node with the animation, as well as the problem.

Visibility

Visibility animation using the MAX visibility controller (see the "eyeball" icon in the MAX track view) is supported. Only values from 0.0 (fully invisible) to 1.0 (fully visible) are supported, and will effect the translucence of all triangles in the given object. Any value lower than 0.0 will be clamped to 0.0, and any value greater than 1.0 will be clamped to 1.0. Visibility has no effect on light or collision objects.

Text "Animation" Events

Text may be "animated" by using the MAX "note track" animation controller. The artist places a note track on the object in track view, then may place key-frames on note track. A text string is then placed in the key-frame which will be exported in the animation as a "text event". When the engine plays an animation that has one of these text events, a message is sent to the animated object that contains the text string. This is used as a free-form method of sending animated commands to an object during run time. The exact meaning of one of these text events is defined by the designer and programmer responsible for the given AI object.

Formatted Structured Events

Much like the text events, the structured events are tracks of animation data that are placed within track view. These events are for formatted data (structured data) where a programmer provides a template in a text file to define the interface. This is much more desirable than a general text event in that the artist sees named fields and checkboxes to interface with instead of a blank text field where they must understand syntax. The ability to do this type of interface is TBD. The unknown portion of it the research required to determine if we overload the MAX note track ability or if we can provide our own animation controller that can handle structured data. This is an "add-on" feature, as the text animation provides all the functionality we require, but does not provide an interface that will limit the user from entering invalid data.

Lights

In addition to having all the above animation characteristics available to them (except visibility), lights have the following extra abilities.

RGB Color

Color may be independently animated for red, green, and blue value.

Intensity Multiplier

The intensity multiplier of a light is a number greater to or equal to 0.0 that is multiplied to the RGB value of the light. This value is set to 1.0 by default.

Range

The range of the light is the distance that it will effect objects in the scene. Note that the engine will actually only allow lights to effect objects that are in the Gem region that the light is in, as well as the Gem regions in the visibility list of the region the light is in regardless of the range of the light.

Falloff Start

The light may also be given a range (lower than the viewable range of the light) where a falloff in light intensity occurs. The effect of the light's intensity will degrade by a TBD formula in the engine for objects in the light that are at a range greater than the falloff. To eliminate this effect, set the falloff equal to the range of the light.

Penumbra Coverage

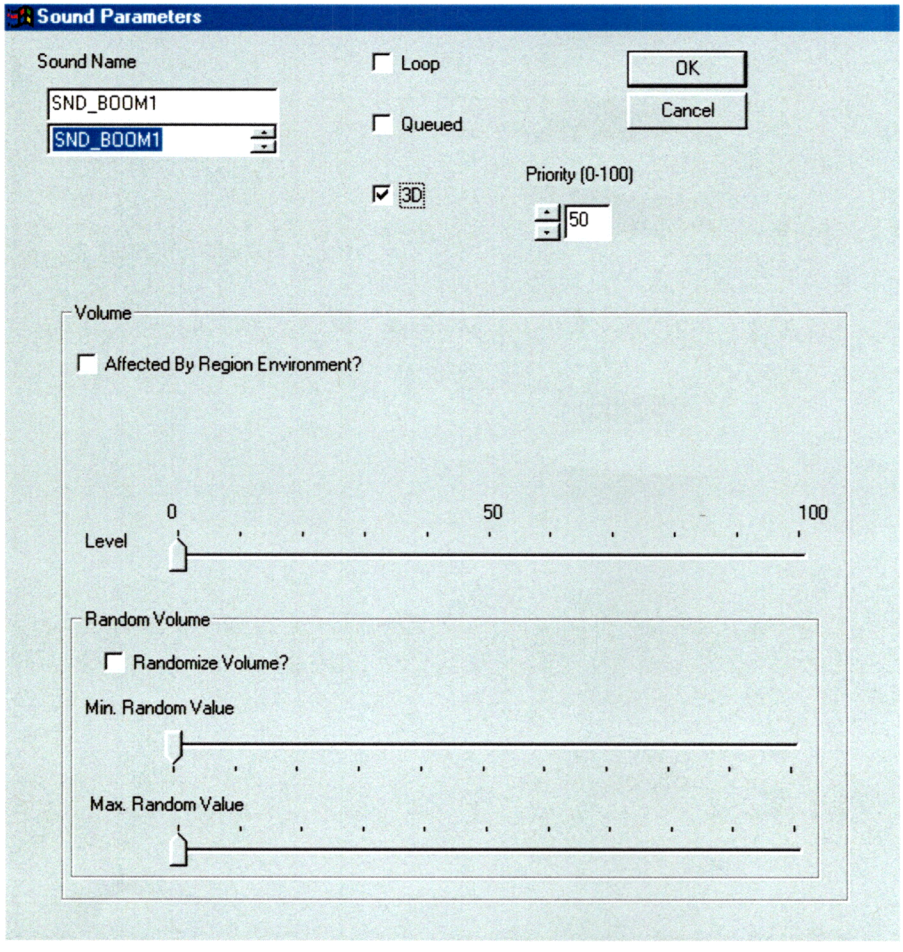
The radius of the penumbra may be animated to produce a larger or smaller "hot spot" for the light. The penumbra is the center "bright spot" of a light, where the umbra is the outer "darker" portion of the light radius. This effect can most easily be seen by illuminating a wall with a flashlight in a dark room. There is a bright spot in the center (penumbra) and a ring of light that is dimmer around the edge (the umbra).

Umbra Coverage

The radius of the light's umbra may be animated to produce a larger or smaller umbra for the light. To completely remove the effects of the umbra and have one light intensity over the entire radius of the light, insure that the umbra is equal to the penumbra. For a definition of the umbra of a light, see penumbra (above).

Sound Events

Sound events are placed on the animation track as a key-frame. The name of the sound shows up in the text for the key-frame. Note that some of the sound data are a set of flags indicating which properties of the sound are affected by regional environment data stored in the scene.



Name

The name of the sound data used to play the sound. Selected from list of available sounds.

Volume

The volume of the sound as a percentage. Defaults to 50%, which indicates to play at the original volume. A horizontal slider control will be used to adjust this. The slider control will have a corresponding check box indicating whether the effect is random (on) or set (off). **Default is off.**

Note: Levels above 50% should be used sparingly and with caution.

If the check box is set to random, there will be two additional sliders indicating the minimum and maximum range of the random effect.

For environment effects, a flag needs to be provided to inhibit environment from affecting a sound volume.

A check box control will be used to choose whether a region environment will (on) affect the sound, or will not (off) affect the sound. **Default is off.**

Pitch

The pitch to play the sound at as a percentage. Defaults to 50%, which indicates to play at the original pitch. A horizontal slider control will be used to adjust this. The slider control will have a corresponding check box indicating whether the effect is random (on) or set (off). **Default is off.**

If the check box is set to random, there will be two additional sliders indicating the minimum and maximum range of the random effect.

For environment effects, a flag needs to be provided to inhibit environment from effecting pitch.

A check box control will be used to choose whether a region environment will (on) affect the sound, or will not (off) affect the sound. **Default is off.**

Reverb

The amount of reverb effect on the sound as a percentage. Defaults to 0%, which indicates to play without any reverb. A horizontal slider control will be used to adjust this. The slider control will have a corresponding check box indicating whether the effect is random (on) or set (off). **Default is off.**

If the check box is set to random, there will be two additional sliders indicating the minimum and maximum range of the random effect.

For environment effects, a flag needs to be provided to inhibit environment from effecting reverb.

A check box control will be used to choose whether a region environment will (on) affect the sound, or will not (off) affect the sound. **Default will be off.**

Environmental effects

Environment effects are qualities of the sound based upon the region of the game the player is in. The following is a list of sound parameters that the designer may manipulate:

- Volume
- Pitch
- Reverb

Each parameter will have a check box indicating whether it will be affected by environmental effects in the region. It will also have a horizontal slider control, allowing adjustment as a percentage. Each slider should default to 50%, which indicates no change to a sound.

See descriptions of Volume, Pitch, and Reverb controls.

Loop

A checkbox indicating if the sound should loop when finished or not. Note that in the engine that when an animation is over that the sound will receive an event to let the sound know it's complete.

If the looping box is checked, each subsequent trigger of the sound by the same animation event should be ignored

Priority

Priority of the sound from 0 to 100. The default is 50. The highest priority is 100. This is used to indicate the importance of the sound when not all sounds may be played. 100 indicates that the sound is non-interruptible.

This control is data entry box with the value to be typed in.

Queue

If the sound may not be played due to it's priority and the current sound load, turning "queue" on will inform the engine that this sound should be played when the sound resources for it are opened up.

A check box control will be used to choose whether the sound should be queued (on) or not (off). The default will be off.

3D

Indicates that the sound is 3D in nature. Defaults to "on". If the sound is a 3D sound, then distance and position from the "listening position" are used to determine characteristics like volume from various speakers.

Plug-in code

One or more plug-ins are required to perform extended editing features as well as exporting a collection of text files that describe a scene and the characters contained in the scene. The features that the plug-in is required to support are listed below.

Gems (Space Partitioning)

Two forms of space partitioning will be used in the tool set to provide higher performance in the game engine. The first is "Gems" (hierarchical bounding volume slabs [KAY86], sited in "Foley & van Dam"), and the second is Octrees. Gems will be implemented as a new object primitive in the plug-in, and will be manually placed in the world by the designer.

Gems perform multiple tasks in the target engine. They are used as a high level trivial rejection scheme for the renderer, as trivial rejection for the collision detection system, a high level AI "Level of Detail" aid, and an animation time-slice inhibitor. By adding extra parameters if required, Gems also are ideal as a world streaming mechanism, providing stream control superior to typical adjacency tests. Gems, while requiring considerable front-end maintenance by the designer, provide an impressive amount of performance benefits that allow the artist/designer to create large, highly detailed environments rich in animation, where the entire world appears to be alive with activity.

Each Gem is contained in a parent bounding Gem to create a hierarchical data base of bounding geometry. In the actual art tool, these are shown as 3D objects that create a wireframe shell (or Gem) around the volume. The artist creates these by simply selecting a group of objects and selecting the Gem tool. The tool will find the smallest volume to contain the objects within, build the Gem, and link all the objects inside to it.

Refer to the Scene manager for more information about Gems.

Model Templates

Model templates are simply the character files that are built in the 3D tool separate from the level, such as a human character, a car, or any other object that will be instanced in the world scene. Model templates may also be built directly into the scene, which is the case for the ground and scenery that is unique. Models are a natural form of geometry compression, as the same template may be instanced in the game at a greatly reduced overhead.

Hierarchical Node Data

Nodes in local 3D coordinates are saved out for the templates. Position, rotation, and scale information is maintained and stored, as well as the name of each node. The data listed in this section is stored with each node.

The node coordinates will be export as a matrix in the form:

```
Matrix
{
   1.0, 0.0, 0.0, 0.0
   0.0, 1.0, 0.0, 0.0
   0.0, 0.0, 1.0, 0.0
   0.0, 0.0, 0.0, 1.0
}
```

Collision spheres

Collision data in the form of radius is stored with the node. The plug-in will create a geometric primitive for a collision sphere that may be scaled to any radius. The primitive will also have an option to be scaled to fully encompass it's local geometry.

Collision sphere data is export as an offset from the node it's attached to and a radius in the form:

```
CollisionSphere
{
   offset 0.0, 0.0, 0.0
   radius 25.234
}
```

LOD minimum vertex set

Level of Detail (LOD) data for the vertices will have a modifier tool built that allows the artist to choose the "minimum level of detail" by selecting the vertices that will never be allowed to be removed. The minimum vertex list will be exported with the geometry data for use in the binary converter. The list will consist of indexes into the vertex list of the selected vertices.

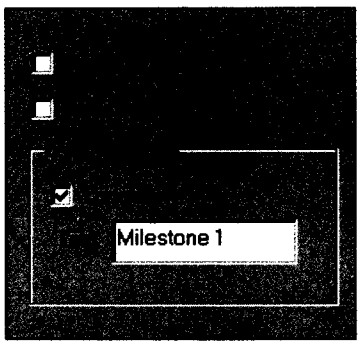
LOD minimum vertex data is export as a list of indices into the vertex array in the form:

```
MinimumVertexIndices
{
   1, 18, 23, 62
}
```

Game specific object data

An interface will be built to allow the artist/designer to set properties for objects. The interface will read the "Object Properties" file shown in the block diagram, where a dynamic interface describing the properties is built to allow the artist/designer to set the properties as required. Default values will be given to all objects applicable. The interface will also allow the user to select all objects in the world, and set properties globally on all applicable objects. Examples of object properties are "billboard/kite" (objects that always face forward), invisible objects (used for detection of crossing a path, or for rails to keep players from falling off the world. See appendix A for a sample Object Properties description file,

the interface created by the file, and a sample of how the exporter includes the properties in the MDF and/or the GDF file.



Object properties are export in an object properties text segment, and due to the nature of the properties not being a standard portion of the export each property is separately enumerated in it's own subsection (enclosed in curly braces). An example of the properties from the dialog above might be as follows:

```
ObjectProperties
{
   FrontFaceing { 0 }
   InvisibleBlocker { 0 }
   TriggerText { "Milestone 1" }
}
```

Blocker Nodes

A plug-in primitive will be built to which will never export, as well as block it's children from exporting. This is an aid for the animator, which will sometimes want objects attached to a character for reference that are not output. An example of this is a human character that can pick up a rocket launcher. The animator would desire to animate a run with the rocket launcher, but not want to export the weapon with the model. By placing a blocker on the hand of the character, and attaching the weapon to it, the "test hierarchy" would never export.

Geometry Data

The geometry data for that requires export is enumerated below.

Vertex coordinates

Exported as floating point triplet data local to the pivot point as follows:

```
Vertex
{
   12.01, -0.23, 5.234,  // Vertex 0
   -13.2, -0.24, 8.23,   // Vertex 1
   45.0, 23.111, 0.0           // Vertex 2
}
```

Vertex normals

Exported as floating point triplet data having a one-to-one correspondence to the vertex data.

```
VertexNormal
{
   1.0, 0.0, 0.0,
   0.7071, 0.7071, 0.0,
   0.5, 0.86, 0.0
}
```

Vertex colors

Exported as floating point triplet data having a one-to-one correspondence to the vertex data. Each of the three values is expressed as a value within a range of 0.0 (no intensity) to 1.0 (full intensity). Vertex colors are export as follows:

```
VertexColor
{
  1.0, 0.56, 0.7, // Color of vertex 0
  0.0, 0.4, 0.0,  // Color of vertex 1
  1.0, 0.8, 1.0   // Color of vertex 2
}
```

UV coordinates

Exported as a pair of floating point 2D coordinates having a one-to-one correspondence to the vertex data. UV values normally range from 0.0 to 1.0 indicating where the vertex maps to it's texture. Values that are greater than 1.0 indicate that the texture is tiled. A U value of 5.0 indicates that the texture is tiled 5 times in the U axis.

```
VertexUV
{
  0.0, 0.2,
  0.0, 0.8,
  1.0, 1.0
}
```

Polygons

Exported as integer triplet data (triangles), where each value is an index into the vertex data to indicate which vertex is associated with the given triangle. Polygon data will be expressed in a clockwise direction, which is used to indicate which direction the polygon faces. Polygons also contain a material id, and a keyword indicating if they are double-sided or not. Polygons are export separately as shown below. Keywords may be added to the polygons using the syntax of "keyword {...}". The binary converter ignores keywords it does not understand.

```
Polygon
{
  VertIndex { 0, 1, 2 }
double_sided { 1 }
}
Polygon
{
  VertIndex { 2, 1, 3}
}
```

Texture Data (Material)

- Texture name
- Attributes

Animation Data

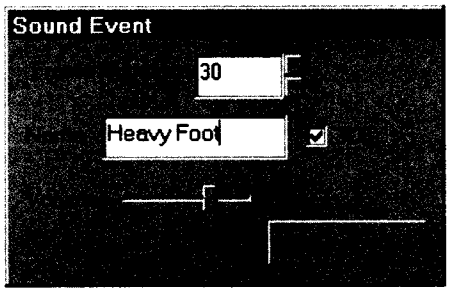
Animation data for all nodes will consist of position, rotation and scale about the local axis of the node. Visibility percentage (translucency) data is also animated. In addition to the normal animated data, the following extra animation data is required.

Light specific Animation Data

Lights shall be able to animate RGB value, Intensity (Multiplier), Range, and falloff. Spotlights shall also be able to animate the Umbra and Penumbra radius.

Event animation

Events are placed in an animation to provide a method for the artist to define when a specific event (such as a footstep sound in an animated walk cycle) is triggered. This gives the artist/designer the freedom to add, remove, edit and control much more of the features that were typically pushed off to the programmers. An example dialog for a simple sound event is shown. The animation events should show up as key-frames in the animation key-framer.



External Data files

AI Templates

AI templates are the method that new objects built for the game may be edited in the tool without polluting the tool with game specific data. Templates are implemented by an ASCII definition similar to a C structure, which defines the data that a designer and an AI programmer require to implement the AI.

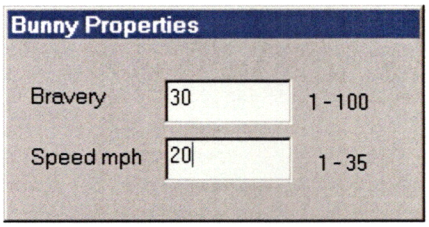
Sample AI Process

In this sample, a designer determines that we need bunnies to be placed at locations in a level. These bunnies should run down holes when a car gets near. After consulting with an AI programmer, it's determined that when a designer places a bunny in the world that 2 attributes should be set for each instance of the bunny. The first attribute is it's bravery (how close the cars will get before it runs), and it's speed (how fast it will run to it's hole). The artist is then given the requirement of producing a bunny with a few animations. The "scripted object" will be called "bunny".

The programmer builds the file "bunny.op" as follows, and then checks in the file to the tool configuration directory:

```
Ai_data "bunny" version 1
{
   int "Bravery"
   {
       default 50
       min 1
       max 100
   }
   float "Speed mph"
   {
       default 20
       min 1
       max 35
   }
}
```

The designer then may place a place-holder object in the level, and set it's "script name" field to "bunny". This will add a properties interface in the level editor dynamically since the script name is the same as one of the AI parameter text files. The designer may now edit the new bunny instance's parameters and export the level with the new bunnies. The designer now has the flexibility to add bunnies anywhere, and set their parameters appropriately. Until the AI programmer completes the next step, bunnies will take on default AI behavior (just sit there and loop through the first animation). Below is a sample of what the bunny property editor will look like.



The AI programmer then uses the AI Template Generator to produce bunny.c, and fills in it's parameter structure with the following:

```
typedef struct
{
   int       bravery;     // 1 - 100
   float     speed;       // 1 - 35mph
} bunnyParam;
```

After compile, the default behavior of the bunny is set... naturally, it will not act upon the new values as the programmer needs to edit the AI portion of the created script c file. But the bunny will show up in the levels if placed in them. The AI programmer may then determine that the designer needs to place a spline down in the level attached to the bunny so that a path to run on may be laid out by the designer, which will give the designer more freedom and free up the programming resource even further.

Note that the text tool will export the following data with instance data in the GDF file (assuming that the designer set the properties to 30 and 20 respectively. It is important that the AI programmer be the one to modify the ai template files, as the data format must match what the actual AI expects to see.

```
ai_properties
{
   int       30
   float     20
}
```

Particle System

Brief overview

The particle system is a crucial part of the Mud, Sweat & Gears game. Particle systems offer us a great deal of bang for the DRAM buck, allowing us to create and manipulate many objects at once without the overhead associated if each object were to have its own separate controller. Particles also add a great deal of color and dynamic feel to a game – bits of sand, smoke and leaves flying around on screen does a lot to enhance the sense of motion and excitement.

The particle system will allow the caller to specify base type of particle (smoke, sparks, sand, etc), the shape and size of the emitter, the number of particles to create, the size and lifetime of the particles, as well as geometry from which to construct the individual particles. Special particle systems will create "autonomous" particles – particles that **do** have their own controller in the Scene Manager, allowing us to use the particle system to create a series of objects on the fly when needed. Furthermore, custom functions can be written to spawn events when the particle dies, or to control the behavior of the particle systems if the default behaviors are insufficient.

Particles will be represented within the Scene Manager in a particle volume, which will have its own collision information and will manage the activities of the individual particles within it. Although we include machinery to allow collisions with individual particles, we anticipate that most particle systems will use collisions based on the volume of the particle system itself.

We anticipate using the particle system for the following types of objects:

- Mud
- Sand
- Dust
- Rocks
- Icicles
- Smoke
- Rain
- Water spray
- Fire
- Leaves
- Logging trucks
- Water haze

Risks – Low

There are two issues – one that we will create too many particles for us to render, and two that we will have trouble finding time to process the particles. Both problems can be solved simply by creating fewer particles. It remains a task for the designers to find the appropriate trade off between speed and appearance.

On particle systems ...

- A particle system consists of an emitter, a particle template, a list of particles, and system parameters.
- Particles do not collide with other particles.
- Particle system may generate objects.

Emitter

Particles are created on the surface of the emitter. Since we want to allow the particles to be created in a variety of areas and with a variety of orientations, we need several different types of emitters. Initially, the following types will be supported, with additions (spline, half cylinder, half sphere) to be added if we find it necessary.

```
typedef enum {
   emSphere,
   emCylinder,
   emCircle,
   emRect,
   emLine
} psEmitterType;
```

Particles that are created on a three dimensional emitter have initial velocity (up to a randomization factor) oriented along the normal outward from the object. Similarly, particles created on a two dimensional emitter also have an initial velocity normal to the surface. However, since it's not clear which direction is "out" of a two dimensional surface, the front face of the surface must be specified when the particle system is created. The normal for a one-dimensional emitter must be specified explicitly.

The position of the emitter defines the position of the particle system, - a particle system with a rectangular emitter, for example, is defined to have its pivot point at the center of that rectangle. The matrix positioning the emitter (and therefore particle system) in the world lives in the particle system's controller (see section on Scene Manager).

We fully parameterize the surface of the emitter. The exact location of the initial positions of the generated particles is determined by generating several random numbers and using those numbers as coordinates within the emitter parameterization.

Canonical Particle

Each particle in the system is generated from a single canonical particle. The canonical particle contains a reference to a standard Scene Manager Template which contains the geometry, animation, and material data for the particle. It also indicates whether the particles are "autonomous" particles - that is, particles which have their own controllers and are only created, but not controlled, but the particle system. (Logging trucks which suddenly appear out of dense wood, or icicles which fall from an overhanging ice shelf are an example of this type of particle). In addition to the geometry template, autonomous particles also contain a reference to the standard controller for the object to be created.

"Controlled" particles (as distinct from autonomous particles) contain additional information for the initial color and alpha value of the particle. Each particle created from the canonical particle receives a copy of these values, and they can be manipulated by the particle system's controller.

```
typedef struct controlledParticleInfo{
   int32      flags;
   char       GeomTemplateName[32];    // Scene manager object template,
contains geometry, animation, and material data
   char       alpha;
   char       r, g, b;
} controlledParticleInfo;

typedef struct autoParticleInfo{
   char       AITemplateName[32];
   char       GeomTemplateName[32];
}

typedef union tagParticleInfo{
   controlledParticleInfo      cpi;
   autonomousParticleInfo      api;
} psParticleInfo;

typedef struct tagCanonicalParticle{
   int type;                          // autonomous or controlled
   psParticleInfo  pi;
} psCanonicalParticle
```

Note

Templates get created during set up as part of the region creation procedure. We plan to create a special "non-visible" region which will include the templates for the canonical particles, thus ensuring that the particle geometry gets created even if the particles themselves do not yet appear in the game.)

Particle System Parameters

The particle system contains many parameters which control the action of the system.

The following structure contains the parameters define the action of the particle system:

```
typedef struct tagParticleSystemParams{
   int32    flags;
   psParticleType  type;        // specifies general action of particles, can
be overriden
   float    breedRate;          // number of particles to create, per second
   float    lifetime;           // standard particle lifetime, seconds
   int32    size;               // size to scale up  (or down)
   float    initialVelocity;    // standard particle initial velocity, normal
to surface
   char     uncertainty[4];     // uncertainty in breeding rate, lifetime, and
velocity magnitude & dir
   float    initialTime;  // start system as if this time had already passed
   (void *)DeathCallback();
   (void *)AICallback(); // non-standard AI callback, if you want
   void *extraData;      // random data, possibly used by AI or death callback
} ParticleSystemParams;
```

The birthRate, lifeTime, and initialVelocity parameters are self explanatory. The four "uncertainty" parameters are related to a randomization factor surrounding these variables. The more uncertain a parameter is, the more variation when the new particle is created. A value of 0 indicates no uncertainty at all. The maximum amount of uncertainty in these parameters is left to the implementation phase - we

presume that the artists are going to want to play with the range of uncertainties until they get something that "feels" nice.

There are two run-time defined callbacks in the system. The first, the Death callback, is called every time a particle dies, and takes as parameters a pointer to the particle system and a pointer to the individual particle that died. The programmer can use this information to trigger a sound, perform an animation, or even create another particle system.

The second callback is the AI callback. There are several default AI/controller functions which control the behavior of the particles based on the type of system – a "sand" system will move the particles out in graceful arcs, a "spark" system will move the particles straight out along their initial velocity vector, and a "smoke" system will elongate the particles and waft them gently upward. However, these default behaviors are not anticipated to fill every need, and therefore the particle system can define its own special controller function. The custom function may be called in one of two ways – if the particle system has a type, which is not "custom", then the AICallback is called in **addition** to the regular controller function. If the particle system's type is custom, the AICallback will be used **instead** of any default controller function.

The particle system also includes flags that define how the system works. These include the following:

```
#define PARTICLE_COLLIDEABLE     0x00000001   /* individual particles collide */
#define PARTICLE_SPRITE          0x00000002   /* always faces toward camera */
#define PARTICLE_VOL_COLLIDEABLE 0x00000008   /* only particle volume collides */
```

There are two types of 'collidable' flags – particle collidable and volume collidable. For a system with the particle collidable flag, individual particles can collide with objects outside the particle system. For a system with the volume collidable flag, the particle volume can collide with other objects, but the collisions of the individual particles are not calculated.

Due to processor constraints, we envision that "particle collidable" systems will be used in only a few select places (if at all.) Volume collidable systems can be used for mud (which will make the car dirty) or water (which may clean the dirt off).

There are no collisions between individual particles or between particle systems, and collisions with particles do not exert force. If you want the particles to act in a dynamic way, they should be created as autonomous particles and given their own controllers.

The full structure of the particle system is given below:

```
typedef struct tagParticleSystem{
  psEmitter                  emitter;
  psParticleSystemParams     params;
  psCanonicalParticle        canonParticle;
  int32                      nParticles;
  psParticleList             *particleList;
  float                      volumeR;        // collision sphere information
  ps3Vector                  volumeOffset;
} psParticleSystem;
```

The Particles

The particles themselves are conceived of as basically "objects lite". They're controlled entirely by the particle system, and are distinguished from one another only by position, orientation, scale, transparency and color. The structure for an individual particle is shown below:

```
typedef struct tagParticle{
   float      birthTime;
   float      deathTime;
   ps3Vector  velocity;
   AnimState  animState;
   uint8      alpha;
   uint8      r,g,b;
   ps4Matrix  lpsMatrix;        // position, rotation, and scale, relative to
particle system
   float      radius;
} psParticle;
```

Note:

A pointer to the geometry is **not** contained in the particle structure, since all geometry is the same and is determined by the canonical particle.)

Particles **can** animate according to animation contained in the particle template. In that case, animation state information is contained in the individual particle.

Interface to the Particle System

The API to the particle system contains functions for creating, destroying, pausing, and modifying a given particle system. Attaching the particle system to the Scene Manager is done through the standard scene manager techniques.

These functions, along with the custom callbacks, should be sufficient to control the particle systems.

Create particle system

```
int32 CreateParticleSystem(psEmitter *emitter, psParticleSystemParams
*params, psCanonicalParticle *particle);
```

Kill particle system and all particles

```
void   KillParticleSystemImmediate( int32 systemID);
```

Kill particle system after last particle dies

```
void   KillParticleSystemSlow(int32 systemID, float lifetime);  /* lifetime
<= 0 means destroy system when all particles are gone */
```

Pause particle system - stops generating particles

```
void  PauseParticleSystem(int32 systemID);
```

Restart particle system

```
void   RestartParticleSystem(int32 systemID);
```

Pause particles - just hold 'em there

```
void   PauseParticles(int32 systemID);
```

Restart particles

```
void   RestartParticles(int32 systemID);
```

Change rate of particle generation

```
float   ChangeParticleBreedRate(int32 systemID, float newRate);
```

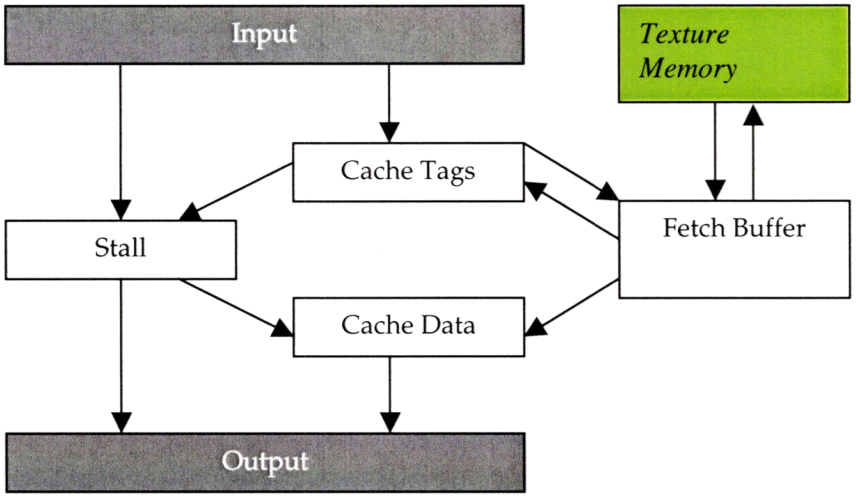
Change particle lifetime of newly created particles

```
float   ChangeParticleLifetime(int32 systemID, float newLifetime);
```

Texture cache

The texture cache will be a standard solution for performance improvement. Texturing wants high bandwidth with small granularity and zero latency. VRAM can provide bandwidth with large granularity and multiple cycle latency. Texture cache minimizes bandwidth demands and provides much granularity of memory's access. Also we plan to combine pre-fetching and caching for best performance.

Simple texture cache structure:



FMA

(also known as Full Motion Animation)

Full Motion Animation (or cinematics) are pre-defined animation sequences that are used at special times during the game. For instance, if the player causes an avalanche to occur, or gets caught up in a tornado, we may want to stop the standard flow of the game and play cinematic sequences.

An FMA "movie" is built up from 5 basic components. Note that the 3 FMA components are AI controllers that are tagged as having FMA AI. The following is a list of the components of an FMA scene.

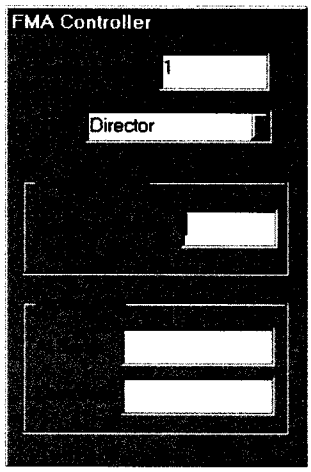
Normal scene data

In addition to the FMA components listed below, an FMA resides in a normal gem or collection of gems (which may or may not be part of the normal scene). Just as any other gem in the scene, these gems may be populated with any type of scenery object or character that is used elsewhere in the scene.

FMA director

Each FMA sequence has one and only one director. The director controls the duration of the "movie", as well as when each camera is given control. The artist defines camera changes with text event animations. To change the camera, a text event of "camera 1" will change the camera to the camera labeled "camera 1". Any number of cameras may be used in an FMA sequence. To define the end of the sequence, the artist has the option of either placing a text event labeled "end", or just setting the length of the animation.

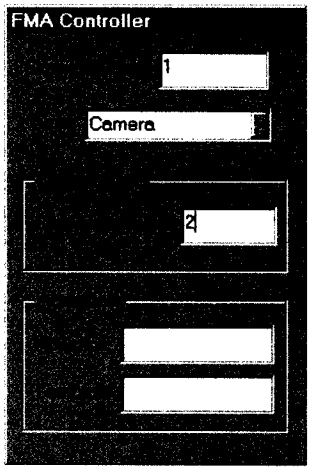
The image to the right is an example of the Calliope™ interface used to define a director for "scene number 1".



FMA cameras

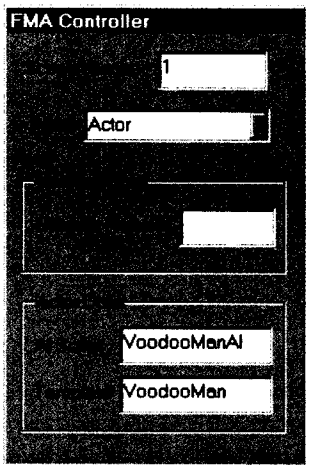
A camera may be placed in the scene and tagged as an FMA camera *(such as "camera 2")*. These cameras may be animated just as any other object. This allows the artist to pan and truck around the scene as appropriate to affect the mood of the sequence.

The diagram to the right illustrates an example of the Calliope™ FMA interface defining the selected object as camera #2 for scene 1.



FMA actors

FMA actors are placeholders for any other AI scripted object. This may be a vehicle, an inhabitant of the Voodoo Village, a light, a particle effect... absolutely any object that has a normal AI controller. The artist tags the placeholder as being an FMA actor, and fills in the name of the AI controller object that will be used in place of the placeholder during the FMA. The picture below is an example of the Calliope™ interface indicating that the selected object is an FMA actor used in scene 1. The AI script that is used (if required) is "VoodooManAI", and the geometry template (with associated animation data) is "VoodooMan".



Golems

The AI controller object that is replaced by the FMA actor is named a golem. This golem is mostly a mindless slave to the FMA actor. The actor instructs the golem how to act, and the golem responds without (except for special cases) even knowing it's happening. This is because when the golem is under FMA control, it's controller is replaced by the FMA actor, so the golem is robbed of it's normal AI timeslice, which is deactivated during the time the FMA is in control.

How FMA actors move golems

The artist lays down splines in the world to define the position and orientation of the placeholder. This spline is actually a normal animation consisting of position, rotation, scale, and text events. If the artist wishes the golem to walk from one side of the room to the other, it will place a spline down as an animation that defines the orientation and velocity to move to the other side of the room. Assuming the golem's normal animations include an animation named "walk", the actor may order the golem to run that animation by using a text event named "walk".

One of the tools required by Calliope™ to let the artist insure that the golem does not slide it's feet during this walk is a spline "speed" control. This is a window that lists the time it takes to get from one key-frame to the next on a spline. The artist must also know the speed that the walk cycle normally takes... then it is easy to match the speed of the walk to the spline by modifying the spline to match the speed required for a proper look.

Animation orders to a golem actually have more than just the name of the animation to play. There are optional arguments to tell the animation to start at a specific key-frame, to loop the animation or to play it once, how many frames to take to tween to this animation from the last one playing, and the speed multiplier to run the animation at (1.0 being normal speed).

FMA Compliancy

Normally, objects under FMA control do not require being FMA compliant. Non-FMA compliant objects have their controllers replaced by the placeholder as described above. For some FMA's, the object will require special AI programming, such as the spotlight of a helicopter may want to stay trained on a specific object (such as the main actor in an FMA). This will require that the normal AI has some control during the sequence. Objects of that type require to be "FMA compliant". Compliant objects will still have their controllers replaced, but their normal controllers will be allowed a time-slice immediately after the actor placeholder. These objects will know that they are in FMA mode, as when the director takes control of the engine, it broadcasts a message on the controller tree that indicates that FMA is starting, and which scene is starting up. Objects that are FMA compliant can handle other FMA messages

attached to the animation track of the FMA actor. These messages start with the character '@', and are a signal that the command is not an animation change, but an order to the compliant controller.

Note that if we find it possible to overload the Calliope™ note controller with a specialized interface that the text formatting of orders will be replaced with a user-friendly interface.

Other objects in the world must be aware that the FMA is occurring, and their actions may either be suspended or modified so that they do not ruin the effect of the FMA. For instance, if the player is at the top of a cliff having just triggered an avalanche, and the FMA sequence is running, we really don't want to allow another car to come along during the sequence and push him/her off the cliff. That would be considered very poor sportsmanship on the part of the game designers.

Starting an FMA

An FMA is started by a message being broadcast in the controller chain that an FMA is starting, and which FMA scene (as in scene 1, 2 or 3) is starting. It is up to the FMA controller assigned to the director for that scene to provide the following services:

- Kill the current message, as the recipient has been found.
- Broadcast a message to the controller tree that the director is starting the FMA.
- Run for the length of the FMA, sending camera change messages when appropriate.
- Broadcast a message to the controller tree to end the FMA upon completion of the FMA.

The actual event to start an FMA is sent by a trigger.

This may be one or more of the following trigger types:

- Scene start
- Scene end
- Timed event
- Collision with a particular object (such as the player's vehicle).

Risks - low

The largest risk is overloading memory with too much animation data, as during the design of an FMA sequence the artist may want to add new animation to existing characters in a scene. For example, the artist may decide that upon entry into the Voodoo Village that on Halloween, (is the PlayStation 2 aware of the date?) the inhabitants of the village gather together and perform a 5 minute "Las Vegas" style dance act, complete with spotlights, musical accompaniment and 200 trained parrots performing acts of acrobatics. Clearly the amount of extra animation data for this is prohibitive. This will be handled by changes in design to fit within memory requirements.

Memory Card

Saving and loading of games is restricted to saving the current tracks and secrets that have been unlocked, as well as the configuration of the player (such as what cars are owned, which options are owned, and how much money the player has). Simply said, the state of the game before the next race is ran is what is saved. Preference options such as sound on/off will also be saved.

Risks – Low

Even in the face of all the unknowns about the NVRAM system that Sony is using, this is not expected to be a high-risk task.

Virtual File System

In order to allow us the maximum flexibility in laying out our files, and to assure that we aren't bitten by a badly written third party driver, we plan to concatenate the files in the game into one large file and use our own TOC to access the actual files within the virtual file.

This system will also be able to compress and decompress data in the virtual file, and also will allow the same file to exist in multiple places in the virtual file. (This second condition is necessary on a CD/DVD, where you need to minimize seek time.)

We will be using a public domain implementation of ZIP for file access and compression.

A person adds files to an existing ZIP file.

The benefit is that ZIP is supported by all platforms and is commonly available.

Risks - Low

This technology is straightforward and has been written many, many times.

Memory Management

DRAM Management

The DRAM memory manager will be a standard pointer based memory manager. We will be writing our own memory manager to get maximum flexibility and to ensure we can fix any bugs that might crop up.

Risks – Medium

The issues here are memory fragmentation and running out of memory. Since we are not using a handle based system, we cannot re-arrange memory to fit the available space, nor can we throw out "lower priority" memory to make room for something more important. If we have problems with fragmentation, we will implement memory pools that will allocate memory in different regions based upon the size or usage of that memory. Problems with sheer memory size will have to be addressed by compression, cutting down on the amount of data in the game, streaming data off the CD, and careful coding.

We will implement a "high water mark" function so that we have an idea of the maximum amount of memory that we've been using.

Fragmentation should be kept at a minimum as the majority of the data in a scene is instanced when the scene is first loaded. The scene manager will insure that all regions and object templates (along with the template object data) are loaded with no holes.

SPU-RAM Management

Based on the same code as the DRAM manager, the SPU-RAM manager handles RAM for the Sound Processing Unit. *This is not a sound-cache.*

VRAM Management

VRAM, a 2 dimensional structure, is considerably more difficult to manage than standard 1-dimensional DRAM. To make things even a bit more difficult, the PS2 allows VRAM "overlays" for textures with fewer than 16 bits for pixel – 2 four-bit textures, for instance, may start at the same location in VRAM but have different byte offsets, so they are interleaved through each other.

We cannot possibly fit all the textures we will use in this game, or even in one level of this game, into 4MB of VRAM. We can allocate certain areas of VRAM to certain types of things – terrain, car textures, shadows, but we'll ultimately be swapping textures in an out from main memory.

Risks – High

This is a traditional ugly problem, and will most likely be one that we rework or add to more than once as we gain experience on the PS2 hardware.

DRAM/SPU-RAM Memory Manager API

Global Function Interfaces:

Memory Initialize

Provides memory manager the location of RAM to manage, and size of that section. Initializes memory.

```
void MEM_Init(u32 start_address, u32 mem_size);
```

Memory Allocation

Interface to allocate memory in the managed region.

```
void *MEM_Malloc(u32 mem_size);
```

Memory Free

Interface to free allocated memory in the managed region.

```
void MEM_Free(void *mem_ptr);
```

Character Allocate

Interface to allocate memory in single byte increments, results will be 32-bit aligned.

```
void *MEM_Calloc(u32 arg_num, u32 arg_size);
```

Debug Memory Management

Control to enable or disable debugging of memory usage.

```
void MEM_setdebug(ON|OFF|DEFAULT|PRINTF|DEBUGFILE);
```

Global Variable Access Interfaces:

Allocated RAM

Returns the amount of RAM, including overhead, that has been allocated.

```
int MEM_RamUsed(void);
```

Available/Free RAM

Returns the amount of RAM that is available for allocating.

```
int MEM_RamAvailable(void);
```

Memory Size

Returns the amount of RAM that is being managed by the memory manager.

```
int MEM_RamSize(void);
```

Memory Top

Returns a pointer to the start of the management memory section.

```
int *MEM_RamTop(void);
```

Geometry Compression

(also known as "fitting it all in")

We need to fit all the geometry for one level into DRAM. Current back of the envelope calculations show that we expect between 2 and 3 times as much geometry data as we can fit into the machine.

Guessing 1 million polygons at 30+ bytes apiece, = 30 + MB.

The artists are hoping to fit the world into fewer polygons (500K), but even so we should plan for some geometry compression.

Possibilities for compression include the use of higher-dimensional representations for cars, and storing vertices as 16-bit fixed point offsets from the center of the object. Other possibilities are to provide traditional file compression algorithms to the geometry and decompress as required.

On higher order surfaces

Cars can be relatively easily and compactly described as Bezier surfaces. This approach would require us to tessellate the surface on the fly, but could potentially save us a good deal of DRAM. Other objects, such as the terrain, are not as good candidates for this type of compression because they have relatively few polygons per Gem, and have inherently sharper edges.

Furthermore, attempting to put a rut-displacement map on a Bezier terrain is probably more work than we have processing power to get into.

Various methods to reduce geometry in DRAM

Filling the world without using geometry:

- Heavy use of particles, dynamic lighting, environmental effects.
- General pervasive use of instanced objects – rendered on the fly from a base of source geometry.
- Use the Tree Populator *(see Scene Generators)* to dramatically reduce source geometry and scalably increase realism.
- Bezier patches, progressive meshes.

Risks – Medium

(TBD)

Instant Replay

Instant replay will be accomplished via saving a stream of user input and starting the race from the beginning in a know state.

Risks – Low/Medium

There are two known risks. The first (the size of the memory card) is what gives this task a Medium status. A lot of input can occur during a race. We will require the ability to store all filtered input as a stream for playback. The second risk is a software discipline risk. We must insure that we can recreate all events based upon random number generators identically each time the same race is ran. This is accomplished by having each random event keep it's own seed, and use methods to pick the seeds so that changes in events such as changing the camera view to not interrupt a predictable pattern during a replay.

Special Effects

Lens effects

Lens flare and lens glare are both attributes that may be assigned to a light using Calliope™. During the render phase, the angle (for all lights except omni lights) and distance of the camera (as well as if any objects occlude the light) are tested. If the light may been seen and is within the proper distance, the proper lens effect is enabled.

Lens glare

Lens glare an artifact of the eye (or lens) where a light source (or a high reflection) gives off a haze. The effect of this haze is an aura centered around the light source that is the color of the light source. Even if a portion of the haze is occluded by an object between it and the camera, the haze is not occluded. This requires that the haze "geometry" is sorted by the render to always be in front of everything else (except maybe status information on the screen).

Lens Flare

Lens flare is actually an artifact of the lens cavity of a camera, and not an effect that may be seen with the naked eye. It seems to work well in games as it has been seen on the sliver screen and TV so much that viewers take it for granted as a valid effect, and if used properly can produce a nice effect.

In a camera, lens flare is produced by particles (dust) in the camera-back and lens cavity that reflect light back toward the lens, which is reflected back to the viewer of the image (or film). Due to the curvature of the lens the effect always appears to be centered in the view. Multiple reflections showing up as a series of circular multi-colored auras that appear to be tied to one spoke that crosses the entire screen (or portions of it depending upon the angle), where the spoke rotates about the center of the screen. Just like lens glare, the auras created by lens flare are always in front of the screen.

Risks – Low

There is relatively no risk in making lens glare work. The only risk (or lack of realism) will be to support the effect of what happens to the lens glare as the source object starts to occlude. You can walk out to your parking lot to see the effect... Stand where you can see the reflection of the sun on a shiny spot of a car until you see the lens glare effect. Now move your position to place a tree or pole in between the glare and your eye. As you come closer to completely occlude the effect it starts to diminish in both size and brightness. This is an "analog" effect vs. just popping on or off. To properly simulate this effect would require casting multiple rays between the camera and the lens glare area to be able to find a percentage of occlusion. While a nice effect, it may not be worth the processing load.

AI Support

The following details software work required to support various characters and effects in MSG, as well as key tasks that although they do not require direct support, they are shown here to give samples of some of the tasks that exist in the project. This is not by any means the complete set of things going on in the game. Due to the method of assigning AI objects and parameters in Calliope™, along with the ability to perform simple animation tasks using a Generic AI controller, and support AI's such as particle emitters, triggers, and swappable objects many of the common AI tasks normally associated with the programming community have been placed in the creative hands of the artists, freeing up the programmers to pursue the perfect tan.

Generic AI controller

The generic AI controller is an engine AI that handles all controllers that are not assigned a controller type (or are given the generic type). The generic AI provides the means for an artist to place an object (or hierarchy of objects) in the world and have them perform simple tasks with no extra programmer support. The artist may assign animations to the objects, which may contain lights, geometry, collision objects... any normally exported object. Given collision geometry, these objects can be tagged to be breakable (or not), and/or to be moveable (or not).

One example is in the mining community there is a train that runs around a track which can not generally be interacted with. The artist may animate that train around the track, make it stop at a water tower, have the water tower fill a tank on the train, have the train start back up and go around the track. All the time smoke could be blowing out of the smoke stack via a particle system, the whistle could blow at the proper times, and the light in the front of the train could be turning back and forth. This complete task can be handled with generic controllers and no programmer intervention.

How would this be done? The artists would make an animation of the train moving around the track, stopping under the water tower. They would set this animation to loop as a property of the controller. They would place a trigger object near the water tower, and tag is as "fire every time object "traincar3" hits it. They would place a collision sphere on "traincar3", which would collide with the trigger when it got close to the water tower. The water tower would be another generic controlled object that was tagged not to play it's one animation until triggered. The trigger would have the water tower as it's target. The whistle blowing would be triggered in the train's animation at the right key-frame to create the sound "whistle" (which would of course have to be a sound asset). The smoke would be the smoke particle emitter which is already in our design to create. The rotating light would just be an animated light that was part of the normal train animation.

Many other types of AI's such as blimps, clouds, or animals may be built using these same techniques.

Particle Effect AI

See the section on particle effects for a description of the actual effect AI. Individual effects required to support programming tasks are given as requirements in each section below.

Waterfalls

Waterfalls shall have translucent effects of multiple layers of water, vertex animation over the surface, scrolling textures, and particle effects for spray at the top of the waterfall and at the bottom.

Support: Vertex animation of water surfaces.

Wooden fences

Wooden fences use swappable objects to replace sections of fence with a particle effect of boards flying when hit hard enough. If hit lightly they will just toggle and spring back. Triggers will be used to activate the collision.

Support: Possible support for detecting collision intensity and translating that to particle emitter parameters.

Barn Doors

Barn doors will swing open an closed if hit lightly. When hit hard, they will swap to a broken version of the door and remain open.

Support: None.

Farm Fencing and Grating

See Barn Doors for grating, and wooden fences for fencing.

Haystacks

Haystacks are swappable objects that swap to an emitter that will make hay fly out. A collision trigger starts it.

Support: Ensure emitters have the quality to "fast forward" n amount of time so that the hay particles are already about the volume of the haystack's surface when the swap first occurs.

Ditch water

Still water that splashes when drove through. Splashes are particles.

Support: Deform water algorithm.

Hay wagon

Rolls out of way when hit.

Support: Insure physics may handle forcing an object to move with impact.

Reeds

Reeds collapse and fall when hit by vehicle.

Risk – Medium

Many individual reeds with their own collision geometry and controllers is extremely unlikely to be done. We are still discussing methods of compromise for this task, such as use the scenery generators and have larger sections of them fall down when hit.

Barricades

Saw-horse style barricades that break when hit.

Support: None

Stream

Moving streams

Support: See section on deformable meshes.

Tractor

Drives around in a pattern.

Support: None

Clouds

Sky may be implemented as translucent domes within translucent domes of cloud textures that animate.

Draw Bridge

When bridge signaled to lower due to a boat on the water hitting a trigger, gates will lower keeping vehicles out. The bridge will open, stay open for some time, and lower again. The gates will then open up. If a vehicle is on the bridge, the gates will not lower.

Support: A bridge AI is required due to requirement that the bridge will not activate if a vehicle is on top of it.

North pole

Breakable pole

Support: None

Snowman

Swappable object from snowman to snow particle.

Support: None

Falling icicles

Particle generator that generates icicle particles that drop to ground, break and disappear. Trigger collision object on ground makes them fall.

Support: None

Breakable Ice crystals

Swappable object swaps to broken crystals.

Support: None

Crumbling glacier

Swapping objects, rigid body animation of swapped object, objects remain.

Support: None

Ice flow

Ice platforms bob up and down, and whole surface waves as one object. Some ice platforms teeter with weight of vehicle.

Support: AI for teetering ice, if surface is not done with node animation from connected nodes below then AI will move objects.

Risk – medium to high

The risk involved is not with the moving ice, but with the AI for the vehicles attempting to find their way across a large constantly changing surface.

Snow storms

Gusting snow and falling snow. The intent is to lower visibility.

Support: AI for snow seems out of scope for a particle generator, so we will most likely do the AI.

Risk – low to medium

The risk is in creating and controlling enough particles to effectively lower visibility as heavy snow would do. We can fall back on fogging if we can not keep the frame-rate up.

Ice water effects

Flowing water (like stream), water falls (just like other water falls, cracking ice effects.

Support: Making cracks in the ice via texture overlays requires AI.

Risk – low

Igloos

Breakable into ice particles. See haystacks, as they work the same.

Generators

Electric generators topple over when hit (triggers and animation), sparks are emitted (triggers and particle effects), lights in area turned off (triggers and animation on lights).

Support: None

Whiskey Barrels

Moveable when hit, breaks if hit hard.

Support: Requires that generic AI handle objects that move when hit, and objects be swappable at a certain hit tolerance.

Deep water

Vehicles float and are carried by current swiftly.

Support: Requires behavior of vehicles in water are modified.

Appendix A

Right Threaded Binary Trees and Tree Traversal

The workhorse of the scene manager is the "Right Threaded Binary Tree", and one function named "treeWalk". Just about every aspect of the scene manager relies on this tree structure and the one walk function. The benefit of this is that the core of the scene manager becomes "bullet proof" very quickly after implementation, as it's used over and over again. This also results in a very small code footprint.

An "RTree" (note that the node is also called an RTree in the code definition) is a binary tree where the nodes point to their parent, child, and their right-hand sibling (Knuth Vol 2). As opposed to a normal binary tree where each node has two children. The RTree is very adept at certain "surgery" tasks, which are relied on heavily in the scene manager. The tree is a depth first tree, which means that the child of a node (and it's children) is processed before the sibling.

Press Start has decided to lower the risk of development and choose C over C++ due to the unknown status of the PlayStation C++ compiler prior to launch. The tree traversal scheme used is definitely an object oriented concept, so the first task is to create a "node object" that has single inheritance characteristics. Below is the definition of an RTree node that is used as the "base class" of all other node types.

```
typedef struct tagRTree
{
   struct tagRTree* pParent;
   struct tagRTree* pSibling;
   struct tagRTree* pChild;
}RTree;
```

Example: Implementing a tree of "color nodes" based upon RTree.

Suppose that we want to make a tree of nodes that contain a color. The new color node definition would look as follows:

```
typedef struct
{
   RTree     tree;
   int color;
}ColorNode;
```

The C language guaranties that when the structure is built that the data in the structure will be placed in the same order as defined in the file [X6J3 committee standard]. This means that casting the ColorNode to type RTree allows us to access the tree linkage information in a standard way, allowing code to link and traverse these trees regardless of the extra data contained within the tree.

"Walking" (or traversing) trees who's nodes are "inherited" from the RTree type can be accomplished with one function prototyped as follows:

```
typedef enum
{
   treeContinue,
   treeHalt,
   treeIgnoreChildren
```

```
}treeRet;

void treeWalk(RTree* pNode, treeRet (*callback)(RTree*,void*), void* pParam);
```

Fortunately, you don't have to look at the prototype of this function very much, as staring at things like this too long can really hurt your head. What this function does is take 3 arguments: pNode is the node to start the traversal at (typically the root of the tree, but does not need to be); "callback" is a pointer to a function that takes an RTree pointer and a void pointer (used like lpParam in a window's callback) and returns a "treeRet" condition. The last argument is the parameter that will be passed into the callback's parameter argument. An example follows to help clear this up.

Example: Traversing a tree of "color nodes" (from above) to count the number of red nodes in the tree.

Assume that we already have a tree built out of "color nodes" (this paper does not cover how to build trees with functions such as AddChild or AddSibling). We first write a callback function that will just check the node pointed to and increment a counter if the node is red.

```
treeRet IsRed(RTree* pNode, void* pParam)
{
   ColorNode* cNode = (ColorNode*)pNode;

   If(cNode->color == RED)
       (int*)pParam += 1;

   return treeContinue;
}
```

Now in the body of a function, we could count the number of red nodes as follows (assume the tree of colorNodes is named pColorRoot):

```
foo()
{
   int colorCount = 0;

   treeWalk(pColorRoot,IsRed,(void*)&colorCount);

   printf("Amount of red nodes = %d\n",colorCount);
}
```

The walk function's behavior is to traverse the tree starting at the given node, and for each node in the tree it calls the callback function, passing the callback the current node being traversed, and the void* parameter that was given in the walk function. The result is that IsRed will be called for each node in the tree, and each time it's called the node will increment the integer value pointed to by the pParam pointer. Upon return from treeWalk, colorCount will contain the count of "red nodes" in the tree.

By inheriting the RTree type, we can make trees of controllers, trees of Gem Bounding objects, templates and instances, and have a general purpose method of tree surgery and traversal.